

# MICROPROCESSORS



## EXPERIMENTS FOR THE MOTOROLA 6800

KERRY URBANIAK



# **EXPERIMENTS IN MICROPROCESSORS**

**FOR THE  
MOTOROLA  
6800**

Kerry Urbaniak



Delmar Publishers Inc.™

I T P™

### **NOTICE TO THE READER**

Publisher does not warrant or guarantee any of the products described herein or perform any independent analysis in connection with any of the product information contained herein. Publisher does not assume, and expressly disclaims, any obligation to obtain and include information other than that provided to it by the manufacturer.

The reader is expressly warned to consider and adopt all safety precautions that might be indicated by the activities described herein and to avoid all potential hazards. By following the instructions contained herein, the reader willingly assumes all risks in connection with such instructions.

The publisher makes no representations or warranties of any kind, including but not limited to, the warranties of fitness for particular purpose or merchantability, nor are any such representations implied with respect to the material set forth herein, and the publisher takes no responsibility with respect to such material. The publisher shall not be liable for any special, consequential or exemplary damages resulting, in whole or in part, from the readers' use of, or reliance upon, this material.

For information, address  
Delmar Publishers Inc.  
3 Columbia Circle  
Box 15015  
Albany, New York 12212-5015

COPYRIGHT © 1994  
BY DELMAR PUBLISHERS INC.

**The trademark ITP is used under license.**

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage or retrieval systems—without written permission of the publisher.

Printed in the United States of America  
Published simultaneously in Canada  
by Nelson Canada,  
a division of The Thompson Corporation

**1 2 3 4 5 6 7 8 9 10 XXX 00 99 98 97 96 95 94**

ISBN: 0-8273-5849-0

Library of Congress Catalog Card Number: 92-21490

# CONTENTS

PREFACE / vii

---

## LAB 1 ENTERING AND EXECUTING A PROGRAM 1

DISCUSSION: Boot Strap Program / 1  
MONITOR PROGRAM / 2  
MNEMONICS / 2  
OPERATION CODE / 3  
6800 ARCHITECTURE / 4  
LAB PROCEDURE / 5  
QUESTIONS / 7  
EXERCISE / 7

---

## LAB 2 DATA TRANSFER 9

DISCUSSION: Data Transfer Instructions / 9  
ADDRESSING MODES / 10  
IMMEDIATE ADDRESSING / 10  
EXTENDED ADDRESSING / 11  
INDEXED ADDRESSING / 12  
LAB PROCEDURE / 12  
QUESTIONS / 15  
EXERCISE / 15

---

## LAB 3 ARITHMETIC INSTRUCTIONS 17

DISCUSSION: Numbers as Information / 17  
DEVELOPING A PROGRAM / 18  
    Step 1: Define the Task / 18  
    Step 2: Lay Out the Steps / 18  
ARITHMETIC INSTRUCTIONS / 19  
LAB PROCEDURE / 24  
QUESTIONS / 29  
EXERCISE / 30

---

## LAB 4 BRANCHING INSTRUCTIONS 31

DISCUSSION: Branching Instructions / 31  
LABELS / 33  
CONDITIONAL BRANCHING / 34

LOOPS / 35  
LAB PROCEDURE / 36  
QUESTIONS / 39  
EXERCISE / 39

---

## LAB 5 SUBROUTINES

41

---

DISCUSSION: Subroutines / 41  
STACK / 42  
KEYBOARD SUBROUTINES / 44  
DISPLAY SUBROUTINES / 44  
LAB PROCEDURE / 44  
QUESTIONS / 47  
EXERCISE / 47

---

## LAB 6 LOGIC, SHIFT, AND ROTATE INSTRUCTIONS

49

---

DISCUSSION: Logic Instructions / 49  
ROTATE AND SHIFT INSTRUCTIONS / 51  
LAB PROCEDURE / 52  
QUESTIONS / 57  
EXERCISE / 58

---

## LAB 7 INPUT/OUTPUT ADDRESSING

59

---

DISCUSSION: Bus Cycle / 59  
DEVICE SELECT PULSE / 60  
DECODING / 62  
OUTPUT / 64  
INPUT / 66  
LAB CIRCUIT / 67  
CIRCUIT OPERATION / 68  
LAB PROCEDURE / 69  
QUESTIONS / 71  
EXERCISE / 71

---

## LAB 8 I/O COUNTER

73

---

DISCUSSION: BCD Instructions / 73  
LAB PROCEDURE / 74  
QUESTIONS / 75  
EXERCISE / 75

---

## LAB 9 INTERRUPTS

77

---

DISCUSSION: Vector Address / 77  
LAB PROCEDURE / 80  
QUESTIONS / 83  
EXERCISE / 83

---

**LAB 10    HARDWARE INTERRUPT 2** **85**

---

DISCUSSION: Interrupt-driven Systems / 85

    Program 10-1: / 86

LAB PROCEDURE / 86

QUESTIONS / 87

EXERCISE / 87

---

**LAB 11    PROGRAMMABLE PERIPHERAL INTERFACE (8255)** **89**

---

DISCUSSION: Programmable Peripheral Interface / 89

CONTROL WORD / 90

LAB PROCEDURE / 93

QUESTIONS / 95

EXERCISE / 95

---

**LAB 12    PARALLEL PRINTER INTERFACE** **97**

---

DISCUSSION: Centronics Printer Interface / 97

LAB PROCEDURE / 99

QUESTIONS / 101

EXERCISE / 101

---

**LAB 13    MEMORY INTERFACING** **103**

---

DISCUSSION: Timing / 103

DECODING / 104

BUS LOADING / 107

LAB PROCEDURE / 109

QUESTIONS / 111

EXERCISE / 111

APPENDIX A    6800 INSTRUCTION SET / 113

APPENDIX B    ADAPTING TO OTHER TRAINERS / 117

APPENDIX C    DATA SHEETS / 119







# PREFACE

This lab manual is an introduction to microprocessor programming and interfacing. Its purpose is to provide an understanding of general microprocessor concepts to someone with no prior microprocessor training or knowledge. It is not intended to teach someone everything there is to know about the Motorola 6800 Microprocessor.

The labs are designed to teach concepts common to all microprocessors, so the knowledge gained can easily be used to further study, or work with, any type of microprocessor. For this reason, only the details of the microprocessor necessary to perform the lab and understand the concepts are covered.

The manual is designed to accompany any introductory text on microprocessors. Since each author covers microprocessor concepts in different sequences, the lab and text sequence may not coincide with each other. The key terms at the beginning of each lab provide information on the topics covered in the lab. These concepts need to be covered prior to performing the lab, either by covering the information from the text in use, or by covering the discussion material in each lab. The information in the discussion section of the labs is sufficient to teach the concepts without the use of a separate text, although this information is somewhat narrow in focus.

The signals used in the interface labs are common to almost any trainer using the Motorola 6800 Microprocessor. Very few (if any) modifications are needed to run the labs on different trainers. Only the details of the monitor program, such as keyboard and display routines, are specific to a particular trainer. Appendix B provides information on adapting the labs to specific trainers.

An understanding of digital electronics is necessary to perform the labs. This includes gate logic, numbering systems, combinational logic, memory circuits, and so on. These concepts should be understood before using this lab manual.

This lab manual uses binary, hexadecimal, and decimal numbering systems. All program addresses, op codes, and numbers used in programs are in hex form. Decimal numbers used in the manual will be followed by a lower-case "d" when the number's form is not obvious.

The sequence of the labs is designed to build on the information in the previous labs. This is especially true with the interface labs, in which the circuit from one lab is modified and used in the following lab. For this reason, it is suggested that the sequence of the labs not be altered.

Appendix C contains the pinout and data sheets for the parts used in the labs. These are common ICs easily obtained from most electronic component retailers.

If the trainer in use has a tape or disk interface, the use of this to save programs should be introduced beginning with Lab 5 or 6. This will save considerable time at entering the op codes with the keyboard for subroutines used by the labs.

To users of Gibson's *Microprocessor Fundamentals: Concepts and Applications*: This lab manual was written to follow Gibson's text as closely as possible. Some concepts in the lab are covered earlier than they are in Gibson's text. This is necessary to provide the information needed to run and understand the lab. As with any text, the information can be covered from the discussion part of the lab, or by modifying the sequence of the text.

The labs correspond to Gibson's chapters as follows:

Labs 1 and 2—Chapter 8

Labs 3 and 4—Chapter 9

Labs 5 and 6—Chapter 10

Labs 7 and 8—Chapter 11

Labs 9 and 10—Chapter 12

Labs 11, 12, 13—Chapter 16

The first seven chapters of this text cover various digital concepts that apply to microprocessors. This information should be understood before running these labs.

I would like to thank my wife Susan for all her efforts in producing this manual. Without her patience, understanding, and word-processing skills, these labs would still be thoughts.

## LAB

# 1

# ENTERING AND EXECUTING A PROGRAM

## OUTLINE

Boot Strap Program  
Monitor Program  
Mnemonics

Operation Code  
6800 Architecture

## KEY TERMS

Add  
Arithmetic Logic Unit (ALU)  
Boot Strap  
Breakpoint  
Machine Code

Mnemonics  
Monitor Program  
Object Code  
Operation Code (Op Code)  
Program Counter (PC)

Prompt  
Source Code  
Register  
RAM  
ROM



Upon completion of this lab, you should know:

1. The use of the trainer's monitor program.
2. The definition and format of mnemonics.
3. The difference between source code and object code.
4. The basic architecture of the Motorola 6800.
5. How the CPU executes a program.

## DISCUSSION

### Boot Strap Program

When power is applied to a microprocessor system, the microprocessor continuously executes instructions it fetches from memory. These instructions, which tell the microprocessor what to do, must be stored in ROM, RAM, or both memory circuits.

The first program a microprocessor runs when power is applied must be stored in non-volatile ROM. This program is often called a **boot strap** or **boot up** program. It will initialize the various components that make up the complete system.

When the boot strap program is finished, the microprocessor starts running a program to interact with the user. This program will set a display to **prompt** the user to enter a command on the keyboard. Note that when the prompt is displayed, the microprocessor is still executing instructions. The CPU is instructed to check the keyboard and determine whether a key is pressed. If no key is pressed, the CPU continues scanning the keyboard until a key is pressed.

## MONITOR PROGRAM

The program that interacts with the user is sometimes called a **monitor program**. The user can direct the monitor program to perform various actions by entering a sequence of 1 or more keystrokes. Some common functions of a monitor program are listed in Table 1-1.

TABLE 1-1

FUNCTION	PURPOSE
Enter	Used to enter hex or binary numbers into the RAM.
Display	Displays the contents of memory locations. The contents normally display as hex numbers.
Go	Tells the CPU to run a program starting at a specified memory address.
Step	Allows the user to single-step through a program 1 instruction at a time.
Register	Used to display or change the contents of a register.

These commands will be used to enter, execute, and examine the results of programs used in the labs. Other functions of monitor programs will be introduced as needed. Because each trainer will vary in the exact format and function of the command, it is important to learn the details of the trainer currently in use.

## MNEMONICS

The instructions the microprocessor fetches from memory are in binary number form. After the CPU decodes the number, it performs the action the number instructed it to do. The exact action the CPU performs will vary significantly from instruction to instruction.

Remembering the exact action resulting from each binary number (instruction) would be very difficult for most people. For this reason, the instructions for a microprocessor are listed in a form called **mnemonics**. It is a shorthand form of representing the action of a particular instruction.

There are 2 parts to most mnemonics:

1. The type of action or operation to perform.
2. The location of the numbers used by the instruction.

In the examples in Table 1-2 following, the first part of the instruction is the operation to be performed, add for ADD A and load Accumulator for LDA B. The ADD A instruction will add a number from a specified memory location to Accumulator A. The result will be placed in Accumulator A. The LDA B instruction will copy a number from a specified memory location and load it into Accumulator B.

TABLE 1-2

MNEMONIC	ACTION	DESCRIPTION
ADD A	ACCA+M→ACCA	Adds the contents of Accumulator A (ACCA) with the contents of memory (m) and places the result in Accumulator A
LDA B	(m)→ACCB	Loads the contents of memory (m) into Accumulator B (ACCB).

A program segment written using mnemonics would look like Program 1-1 in Table 1-3.

TABLE 1-3  
Program 1-1

MNEMONIC	DESCRIPTION
LDA A,N (IMM)	Load Accumulator A with data using immediate addressing mode.
ADD A,N (IMM)	Add to ACCA data using immediate addressing mode.
STA A,N (DIR)	Store the contents of ACCA in memory using direct addressing mode.

The addressing mode in parentheses in Program 1-1 identifies how the address of the data in memory is determined. Addressing modes are covered in detail in Lab 2.

## OPERATION CODE

After a program is written using mnemonics, each instruction must be converted to the binary number that tells the CPU to execute the action of the particular instruction. These numbers must then be stored in RAM so the CPU can fetch them for decoding. The numbers, which generate the action of the mnemonic, are called the **operation code** or **op code**. The op code is sometimes referred to as the **machine code**. The op codes in hex form for the mnemonics in Program 1-1 are listed in Table 1-4 as they would be stored in memory.

TABLE 1-4

MEMORY ADDRESS	OP CODE	INSTRUCTION
1000	86	LDA A,80 (IMM)
1001	80	Data loaded into ACCA
1002	8B	ADD A,37 (IMM)
1003	37	Data added to ACCA
1004	97	STA A,F0 (Dir)
1005	F0	Memory address

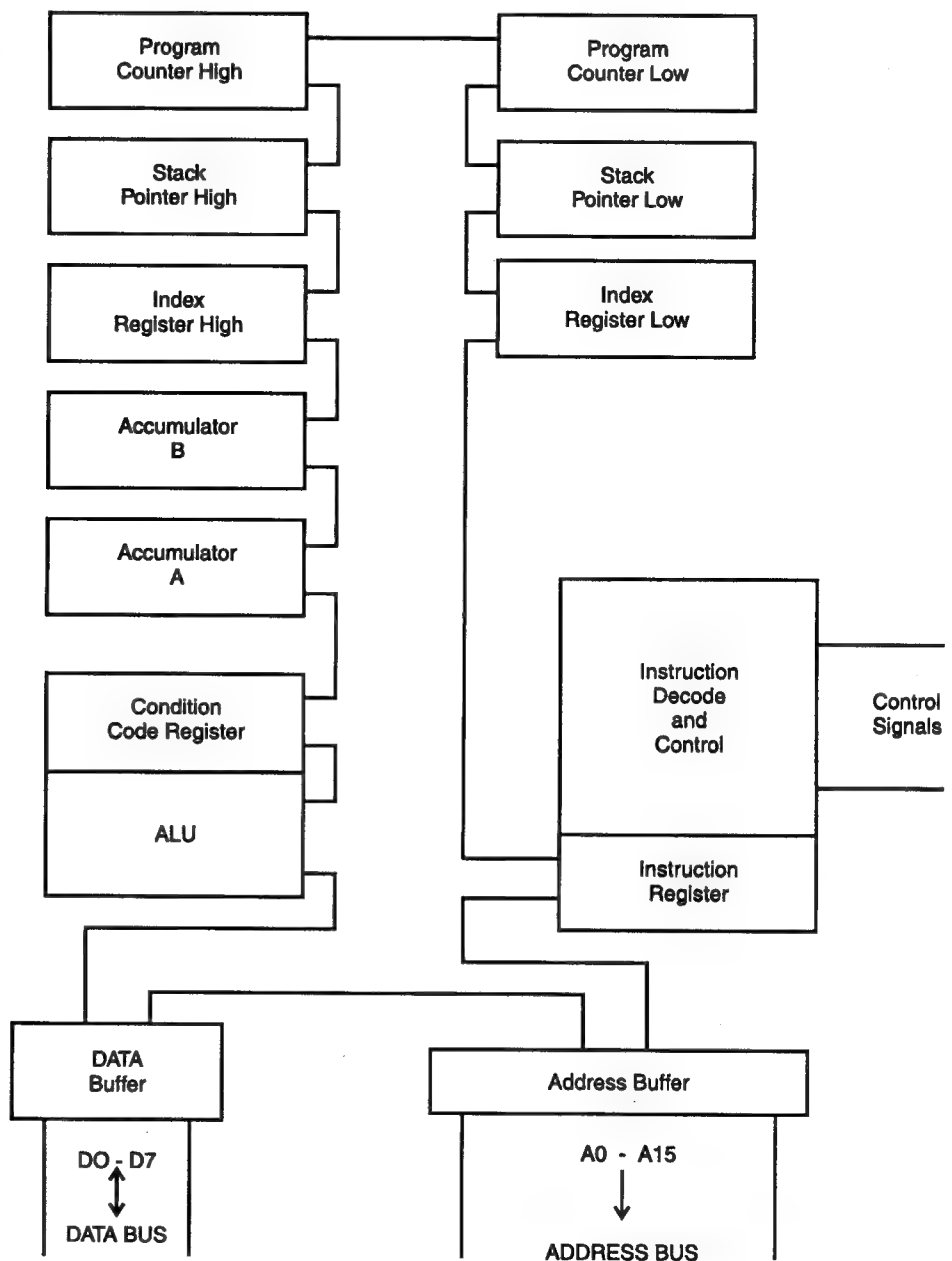
A program written with mnemonics is called the **source code**. A person or another program must read each mnemonic and interpret it into the

respective operation code before a microprocessor can execute the instruction. After the source program is interpreted into op codes, the resulting sequence of numbers is called the **object code**. The object code is a program consisting of the actual numbers (op codes) that tell the CPU what action to take. The object code can then be stored in memory, from which the CPU can fetch and execute each op code. The object code is the only program a microprocessor can run.

## 6800 ARCHITECTURE

Programming a microprocessor with individual instructions requires a knowledge of the registers inside the CPU. A simplified block diagram of a 6800 microprocessor is shown in Figure 1-1.

**FIGURE 1-1 Simplified 6800 Architecture**



Data and op codes are moved between the CPU and memory over the 8 bit data bus. The control section routes op codes to the instruction register for decoding, and data bytes to and from the internal registers.

Addresses for op code fetches are generated by placing the contents of the program counter (PC) on the address bus. The PC is a 16 bit register automatically incremented by 1 after each use. This keeps the PC always "pointing" to the address of the next instruction.

The stack pointer and index register are also used for memory addresses. Their exact use will be covered in later labs. The index register can also be used to store temporary data.

Two 8 bit accumulators, called ACCA and ACCB, hold the data numbers used by the Arithmetic Logic Unit (ALU). The accumulators also hold the results produced by the ALU.

The condition code register is a set of flip-flops that indicates the results of the ALU. The use of the condition code register will be covered over several labs.

---

## LAB PROCEDURE

1. Enter the machine code for the following program into RAM using the Enter command. Note that the actual address for the PC register will vary between trainers. If the address in the PC starts somewhere other than 0000, the address of the instruction (op code) will be the starting PC plus the address listed in the following.

ADDRESS	CONTENTS	COMMENTS
0000	86	LDA A (IMM)
0001	80	Data loaded into ACCA
0002	8B	ADD A (IMM)
0003	37	Data added to ACCA
0004	97	STA A (Dir)
0005	70	Memory address

2. Use the display command to display the contents of memory locations 0000 through 0005. The 6 numbers entered in exercise 1 should be displayed. If not, use the enter command to correct them.
3. Use the step command to single step through each instruction entered in exercise 1. After executing each instruction, record the contents of the requested registers and memory addresses. The PC must have the address of the first instruction (0000) before the first single step. If it does not, use the register command to set it.

Start PC \_\_\_\_\_ ACCA \_\_\_\_\_  
 0070 \_\_\_\_\_

### Instruction 1:

PC \_\_\_\_\_ ACCA \_\_\_\_\_

Instruction 1 will load the number 80 into Accumulator A. The PC will now point to the next instruction, (Add A, AL), at address 0002.

### Instruction 2:

PC \_\_\_\_\_ ACCA \_\_\_\_\_

The second instruction will add the number 37 to the contents of ACCA (80). The result will be stored in ACCA. The PC is now pointing to the third instruction at 0004 (STA A).

**Instruction 3:**

PC \_\_\_\_\_ ACCA \_\_\_\_\_  
 0070 \_\_\_\_\_

The third instruction will store the contents of ACCA in memory location 0070. The PC now contains the address 0006, the address where the next instruction should be located. If you were to single step a fourth time, the CPU would fetch the number at 0006 as a fourth instruction. It would execute the action of that number as if it were an op code that is part of the program.

4. Use the Register command to change the contents of ACCA back to 00.
5. Use the Go command to run the program at full speed. Be sure to include the starting address (0000) and the ending address G=0000 0006.

The first address (0000) is the starting address. The second address is known as a **breakpoint**. A breakpoint stops the CPU before it executes the instruction at the specified address. This feature is very useful at debugging programs.

The monitor program includes a function used to set breakpoints. As with the other monitor commands, the exact format will vary between trainers.

6. Record the contents of the following registers.

PC \_\_\_\_\_ ACCA \_\_\_\_\_  
 0070 \_\_\_\_\_

---



# **LAB 1 ENTERING AND EXECUTING A PROGRAM**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

## **QUESTIONS**

---

1. How many memory locations were needed to store the 3 instructions?
2. Explain the changes that occurred in the PC when single stepping.
3. The number that tells the microprocessor what to do is called a(n) \_\_\_\_\_.
4. What address would the CPU place on the address bus if the program were single stepped a fourth time?
5. What is the difference between source code and object code?
6. What was the difference between running the program at full speed and single stepping?
7. Why is a breakpoint necessary when running a program at full speed?

## **EXERCISE**

---

1. Practice using the different commands of the monitor program for your trainer.



# LAB 2

# DATA TRANSFER

## OUTLINE

Data Transfer Instructions  
Addressing Modes  
Immediate Addressing

Extended Addressing  
Indexed Addressing

## KEY TERMS

Addressing Mode  
Data Transfer  
Direct Addressing

Extended Addressing  
Immediate Addressing

Indexed Addressing  
Operands



Upon completion of this lab, you should know:

1. How a microprocessor addresses an operand.
2. The use of four addressing modes: immediate, direct, extended, and indexed.
3. The result of data transfer instructions.
4. How to manually assemble a program.

## DISCUSSION

### Data Transfer Instructions

**Data transfer** is the process of copying binary numbers (data) from one location to another. These locations can be registers inside the CPU, specified memory locations, or from memory as part of a program.

Moving numbers to and from different locations is perhaps the most common instruction used in programs.

Numbers (data) stored in memory, for example, must be moved to the CPU before they can be examined or manipulated by a program. When numbers are used in arithmetic or logic operations, one or both must be in the accumulators. If the result is needed later in the program, it must be saved in memory before the ALU is used again. How and where numbers are moved are important parts of most programs.

## ADDRESSING MODES

The numbers (data) manipulated by instructions are called **operands**. How the instruction refers to the location of the operand is called the **addressing mode**. There are many different methods of addressing data, which vary among different microprocessors; however, most microprocessors have addressing modes similar to (if not the same as) other microprocessors. The addressing modes covered in this lab are common to most processors.

Table 2-1 lists 4 addressing modes for some of the data transfer instructions.

TABLE 2-1

		ADDRESSING MODES			
		IMM	DIR	IND	EXT
OPERATION	MNEM	op ~ =	op ~ =	op ~ =	op ~ =
Load acmltr	LDA A	86 2 2	96 3 2	A6 5 2	86 4 3
	LDA B	C6 2 2	D6 3 2	E6 5 2	F6 4 3
Store acmltr	STA A		97 4 2	A7 6 2	B7 5 3
	STA B		D7 4 2	E7 6 2	F7 5 3
Load index register	LDX	CE 3 3	DE 4 2	EE 6 2	FE 5 3
Load stack pointer	LDS	8E 3 3	9E 4 2	AE 6 2	BE 5 3
Store index register	STX		DF 5 2	EF 7 2	FF 6 3
Store stack pointer	STS		9F 5 2	AF 7 2	BF 6 3

The op code (op), number of machine cycles required for executing (~), and number of bytes in the instruction (=) for each addressing mode are listed under their respective headings. For example, the op code for LDA A (IND) is A6. It requires 5 machine cycles to execute. The instruction consists of 2 bytes, the op code followed by the offset, which are added to the IX register.

## IMMEDIATE ADDRESSING

When the operand is included as part of the instruction, it is called **immediate addressing**. The operand is in the memory address *immediately* following the op code of the instruction. The number used as the data byte cannot normally be changed; it is a permanent part of the program. An example follows.

MNEMONIC	ACTION	DESCRIPTION
LDA X, N (IMM)	ACCX ← (m)	Loads the number in the memory address following the op code, N, into the specified accumulator (X=A or B).

This is a 2 byte instruction. The first byte is the operation code (in the example, load accumulator), and the second byte is the operand, the number loaded into the specified accumulator. An example follows.

MEMORY	CONTENTS	DESCRIPTION
02AB	86	Op code for LDA A,N (IMM)
02AC	85	number loaded into ACCA.

The index register and stack pointer can also be loaded using immediate addressing. Because they are 16 bit (2 byte) registers, the two locations following the op code in program memory would contain the operand. The first number following the op code would be loaded into the high byte of the register. The third byte of the instruction would contain the number loaded into the low byte of the register. An example follows.

MNEMONIC	ACTION	DESCRIPTION
LDX	IXH $\leftarrow$ (m) IXL $\leftarrow$ (m+1)	Loads the number in the memory address following the op code into IXH, and the number in the next memory address into IXL.

MEMORY	CONTENTS	DESCRIPTION
1A70	C3	Op code for LDX (IMM)
1A71	20	Number loaded into IX high
1A72	FF	Number loaded into IX low

After this instruction is executed, the IX register will contain the number 20FF.

When **direct addressing** is used, the address of the memory location is contained in the second byte of the instruction. Direct addressing can access only the first 256d bytes of memory addresses, 0000 through 00FF. The upper 8 bits of the address bus will all be low. This is the fastest addressing mode for moving operands between the CPU and memory.

MNEMONIC	ACTION	DESCRIPTION
STA X	(M) $\leftarrow$ ACCX	Store the contents of ACCX in the memory location directly addressed by the byte following the op code.

MEMORY	CONTENTS	DESCRIPTION
21B6	97	Op code for STA A (Dir).
21B7	A7	Direct memory address.

This instruction will store the contents of ACCA into memory location 00A7.

## EXTENDED ADDRESSING

**Extended addressing** is very similar to direct addressing in that the memory address is specified in the instruction. Extended addressing uses 2 bytes to specify the address, instead of 1 as in direct addressing. This allows extended addressing to access any of the 64K memory addresses. The first byte following the op code is the high byte of the memory address. The next byte (third byte of the instruction) is the low byte of the memory address.

MNEMONIC	ACTION	DESCRIPTION
STX (EXT)	(m) $\leftarrow$ IXH	Store IXH in the address specified by the program.

MNEMONIC	ACTION	DESCRIPTION
	$(m+1) \leftarrow IXL$	Store IXL in the address specified by the program + 1.
MEMORY	CONTENTS	DESCRIPTION
F00F	FE	Op code for STX (EXT).
F010	1E	High byte of memory address.
F011	00	Low byte of memory address.

This instruction will store the high byte of the IX register in memory location 1E00. The low byte of IX will be stored in 1E01. If extended addressing is used with an 8 bit operand, only one byte would be stored in, or loaded from, memory.

## INDEXED ADDRESSING

When **indexed addressing** is used, the byte following the op code is added to the low byte of the index register. If a carry is produced, it is added to the high byte of IX. The modified address is then used as the memory address for the operand. The modified address is held in a temporary register, so the value in the index register remains unchanged.

Before indexed addressing can be used, the IX register must be loaded with the base address. The byte following the op code (the number added to IX) can be considered as an offset from the base value loaded in IX.

MNEMONIC	ACTION	DESCRIPTION
LDA X (IND)	$ACCX \leftarrow (m)$	Load ACCX with the number in the memory location addressed by adding the IX register with the byte following the op code.
MEMORY	CONTENTS	DESCRIPTION
2007	CE	Op code for LDX (IMM).
2008	20	Number loaded into IXH.
2009	00	Number loaded into IXL.
200A	E6	Op code for LDA B (IND).
200B	0A	Offset added to IX.

The first instruction, LDX (IMM), loads the IX register with 2000, using immediate addressing. The second instruction, LDA B (IND), loads the number from memory address 200A (IX + offset) into ACCB.

## LAB PROCEDURE

### Program 2-1:

1. Enter the following program:

Addr	Object Code	Mnemonic
0000	86	LDA A (IMM)
0001	B5	Data byte
0002	97	STA A (Dir)
0003	70	Address byte

Addr	Object Code	Mnemonic
0004	B7	STA A (EXT)
0005	00	High byte of address
0006	71	Low byte of address

2. After you have entered the program, single step through the 3 instructions, and observe and record the contents of the requested registers for each instruction.

**Instruction 1:**

ACCA \_\_\_\_\_

PC \_\_\_\_\_

**Instruction 2:**

ACCA \_\_\_\_\_

PC \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_

PC \_\_\_\_\_

3. Examine memory location 0070 and 0071 and record the contents.

0070 \_\_\_\_\_

0071 \_\_\_\_\_

4. Now add to the program the following segment:

Addr	Object Code	Mnemonic
0007	CE	LDX (IMM)
0008	00	High data byte
0009	70	Low data byte
000A	A7	STA (IND)
000B	02	Offset byte

5. Single step through the first 3 instructions again. Starting with the new instruction at 0007, observe and record the requested registers.

**Instruction 4:**

ACCA \_\_\_\_\_

PC \_\_\_\_\_

**Instruction 5:**

ACCA \_\_\_\_\_

PC \_\_\_\_\_

6. Examine memory location 0071 and 0072 and record the contents of those memory locations.

0071 \_\_\_\_\_ 0072 \_\_\_\_\_

7. Explain the results obtained for each of the 5 instructions.

**Program 2-2:**

1. Load the following program.

Addr	Object Code	Mnemonic
0000	86	LDA A (IMM)
0001	A5	Data byte
0002	CE	LDX (IMM)
0003	00	High data byte
0004	60	Low data byte
0005	97	STA A (Dir)
0006	70	Address byte
0007	06	LDA B (Dir)
0008	70	Address byte
0009	A7	STA A (IND)
000A	00	Offset byte
000B	E7	STA B (IND)
000C	01	Offset byte
000D	B7	STA A (EXT)
000E	00	High address byte
000F	62	Low address byte

2. Single step through the program and answer the following questions.

- a. What data is loaded into ACCA at line 0000, and from where?

Data Byte \_\_\_\_\_ From \_\_\_\_\_

- b. The instruction at line 0005 loads the data byte \_\_\_\_\_ into memory location \_\_\_\_\_.
- c. The instruction at line 0009 loads the data byte \_\_\_\_\_ into memory location \_\_\_\_\_.
- d. The instruction at line 000B loads the data byte \_\_\_\_\_ into memory location \_\_\_\_\_.
-



## LAB 2 DATA TRANSFER

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

1. When the accumulator contents are stored in memory, does the number in the accumulator change?
2. How is the memory address determined when using indexed addressing?
3. When examining the registers during the single step mode, the Instruction Pointer (IP) contains the address of which instruction?
4. There are 7 instructions in the last program. List the line number of each instruction and state which instruction addressing mode was used.

### EXERCISE

---

1. Manually assemble the following source code into object code, using Table 2-1. Enter the object code into memory and execute the program.

**Source code**

```
LDA A,77 (IMM)
LDX,0050 (IMM)
STA A,00 (IND)
LDA A,00 (IMM)
LDA A,0050 (EXT)
STA A,70 (DIR)
```



## LAB

# 3

# ARITHMETIC INSTRUCTIONS

## OUTLINE

Numbers as Information  
Developing a Program

Arithmetic Instructions

## KEY TERMS

Auxiliary Carry  
Carry Flag (CF)  
Compare (CMP)  
Decrement (DEC)

Flowcharting  
Half Carry (HF)  
Implied (IMP)  
Increment (INC)

Machine Language Program  
Sign Flag (NF)  
Zero Flag (ZF)



Upon completion of this lab, you should know:

1. How to develop a program.
2. The use of arithmetic instructions.
3. How arithmetic instructions affect the flags.

## DISCUSSION

### Numbers as Information

When programming on a machine level, think in terms of how a microprocessor operates. Because a microprocessor can process only numbers, we must learn how to represent information as numbers and manipulate the numbers to produce desired results. Look at an 8 bit binary number and see how it can represent several different types of information. The binary number 0011 0101 can be any of the following.

The binary equivalent of 53 base 10, which can represent temperature, speed, and so on.

The ASCII code for the character "5" (35 Hex).

Two B C D digits (3 and 5).

The position code of a key on a keyboard (row 3, column 5).

Eight binary digits, each representing the status of a door (0=closed, 1=open).

As you can see, a number can represent any kind of information. What it represents depends on how the number was obtained (keyboard, A/D converter, and so on) and how it is defined in the program.

Writing programs to produce useful results requires learning ways of manipulating numbers so that the information they represent can be identified. Most programs will use arithmetic and/or logic instructions to manipulate the numbers representing the data. This lab will cover some of the arithmetic instructions available in the 6800 instruction set.

## DEVELOPING A PROGRAM

---

The other difficult part of writing a program in assembly is the amount of detail that must be considered. The result of executing an instruction is usually quite simple, so a program that accomplishes a complicated task requires many small, detailed steps. An organized approach to developing programs helps reduce the time and effort needed to take a program from the conceptual stage to the final working form. There are basically 4 steps involved in developing a program.

1. Define the task the program is to accomplish.
2. Lay out the steps needed to perform the task.
3. Put together the instructions to accomplish each step identified in step 2.
4. Debug the program.

To illustrate this procedure, a program will be developed to change 2 hex numbers, stored as 1 byte in memory, to the correct ASCII codes and store them in 2 different memory locations. The complete program will be developed over the next several labs.

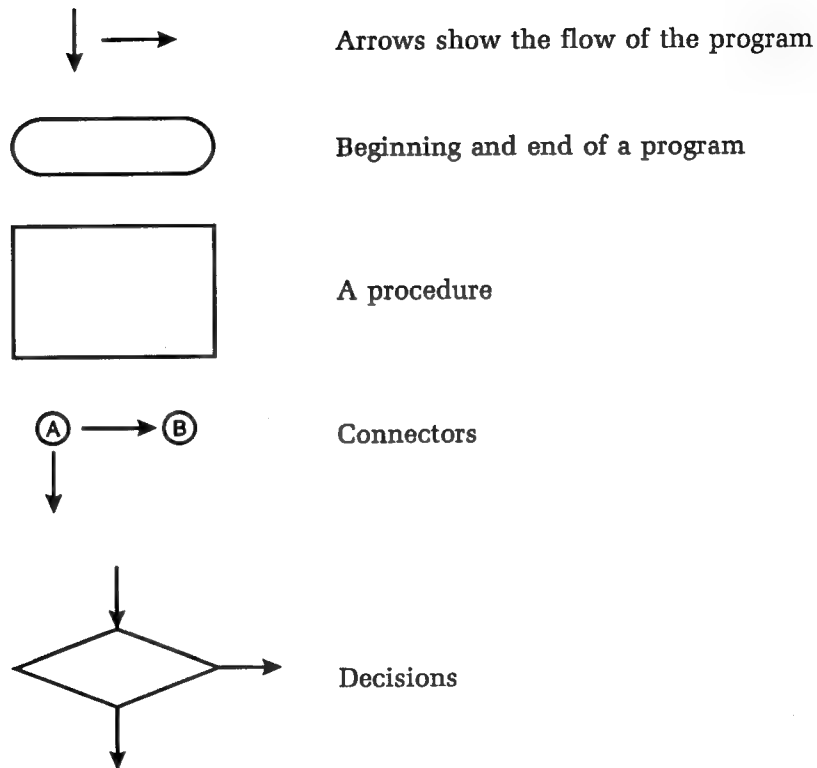
### Step 1: Define the Task

As we are working with ASCII codes, we should first examine the ASCII chart. The decimal digits 0–9 have a code of 30–39. The additional hex digits A–F have a code of 41–46. From this information, we can see that adding 30 to a hex digit will convert it to ASCII if it is 0–9. However, to convert A–F, 37 must be added to the hex number. From this we can see that the number must be identified as either 0–9 or A–F so that this correct conversion number can be added to the hex digits.

The preceding thoughts will help develop the actual conversion process, but the state of the number before conversion must be considered. The number requiring conversion is actually 2 hex digits stored as 1 byte in memory. Because the microprocessor can do only one thing at a time, the program must separate the digits (nibbles), save 1, and convert the other. It must then save the first converted ASCII code, restore the second digit, convert it, then put both ASCII bytes into different locations (in the correct high-low order).

### Step 2: Lay Out the Steps

Step 2 requires us to logically lay out the steps needed to accomplish the task defined in Step 1. This is called **flowcharting**. Each step is called a *block*; special symbols are used to represent types of blocks. Only a few of the many symbols will be shown in Figure 3-1a as others are used for specific equipment and functions.

**FIGURE 3-1a Flowchart Symbols**

The flowchart for the HEX to ASCII conversion program is shown in Figure 3-1b.

After a flowchart is completed, it can be examined to see if anything has been left out or whether it can be modified to make it more efficient. When examining the HEX to ASCII flowchart, notice the actual conversion process. The 3 blocks, which check for less than A, Add 30, or Add 37, are the same for both high and low nibbles. Whenever a series of blocks is repeated, it is more efficient to write 1 sequence of instructions and use the same routine at each point in the program where that process is needed. The modification to the flowchart is shown in Figure 3-2.

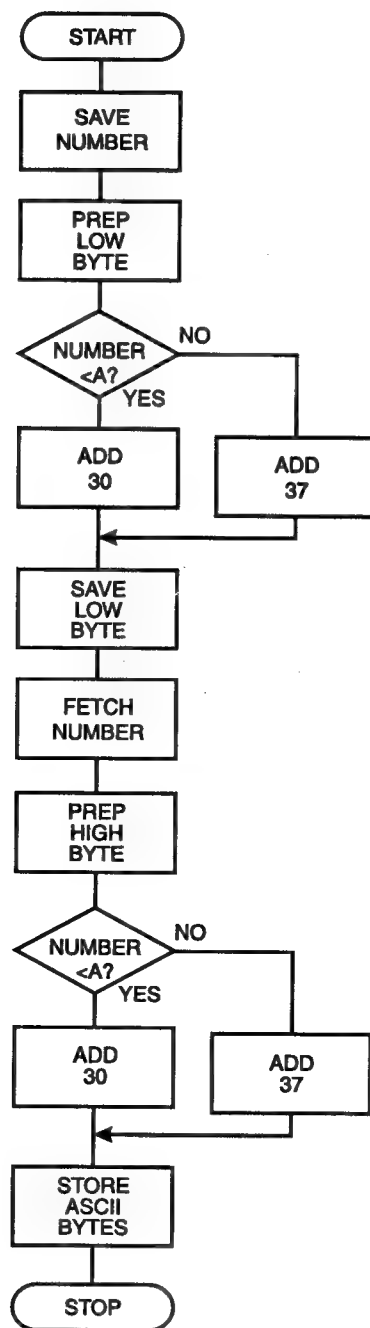
The subroutine conversion is shown in Figure 3-3.

Many people try to skip flowcharting and for small, simple programs, get away with it. But as programs get more complicated, any time saved by skipping this step will be lost many times over when writing the program using instructions. Try to develop good habits now—it will pay off later!

## ARITHMETIC INSTRUCTIONS

The arithmetic instructions in Table 3-1 add an operand from memory to 1 of the accumulators. The result of the addition will be placed in the specified accumulator. For example, the instruction `ADD A,F0 (Dir)` will cause the microprocessor to add the number in memory location 00F0 to the number in ACCA. After the instruction has been executed, ACCA will contain the sum of the 2 numbers, and memory location 00F0 will remain unchanged.

At the right side of Table 3-1, the condition codes affected by the instructions have been added. The condition codes, often called flags, are flip-flops (either set or reset) used to indicate the result of certain instructions. The **Carry Flag (CF)** is used to indicate a carry from the high order bit, 07, or a borrow into the high bit resulting from an add or subtract instruction. If,

**FIGURE 3-1b Sample Flowchart**

The starting point

Save the number so the high nibble can be converted after the low nibble

Make the upper 4 bits 0, AL will have the form 0000XXXX

Determine if the digit is 0-9, or A-F

Add the correct conversion factor

Save the converted low digit

Get the number so the high nibble can be converted

The number must be in the form of 0000XXXX before adding the conversion factor

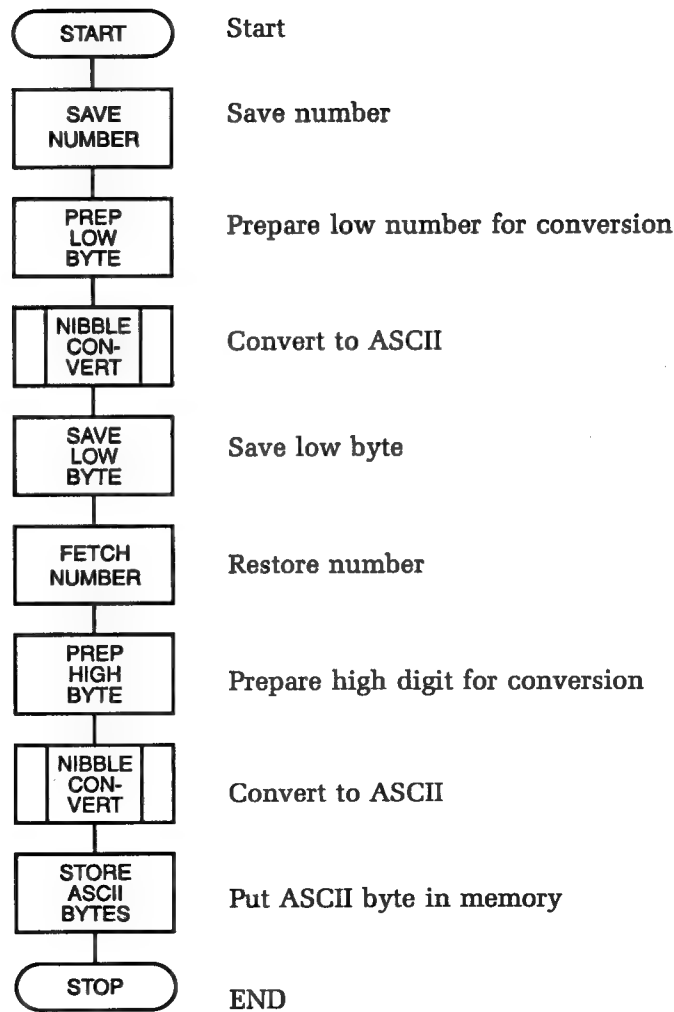
Determine if digit is 0-9 or A-F

Add conversion factor

Put ASCII bytes in memory as final step

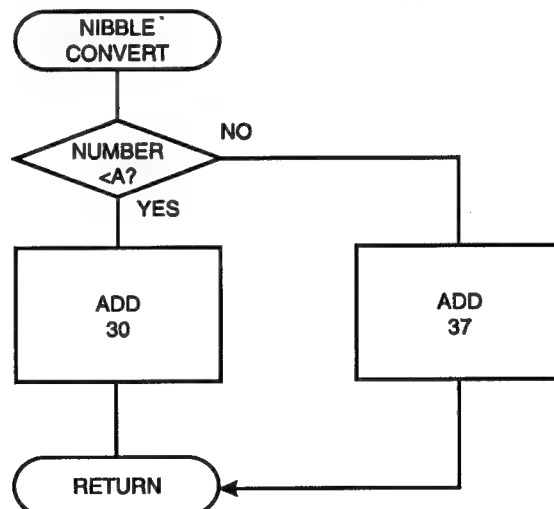
for example, the instruction ABA (add ACCB to ACCA) is executed and the registers have the numbers B8 in ACCA and 90 in ACCB:

		HEX
	D7 D0	
ACCA	10010000	B8
	+	
ACCB	10111000	90
	-----	
result in		
ACCA	01001000	48
	+ carry of 1	

**FIGURE 3-2 Modified Flowchart**

the result would be 48 with a carry of 1, the hex number 148. In this case, the carry flag (CF) would be set to a 1. The carry flag is D0 in the condition code register.

The bit D5 in the condition code register is the **Auxiliary Carry** or **Half Carry (HF)**. It is set if there is a carry from the low nibble to the high nibble

**FIGURE 3-3 Subroutine Conversion**

**TABLE 3-1**  
**Arithmetic Instructions**

OPERATION	MNEM	ADDRESSING MODES					Cond code res					
		IMM	DIR	IND	EXT	IMP	5	4	3	2	1	0
		op ~ =	op ~ =	op ~ =	op ~ =	op ~ =	H	I	N	Z	V	C
ADD	ADD A	8B 2 2	9B 3 2	A8 5 2	BB 4 3		↑	•	↑	↑	↑	↑
	ADD B	CB 2 2	DB 3 2	E8 5 2	FB 4 3		↑	•	↑	↑	↑	↑
ADD Acmltrs	ABA					1B 2 1	↑	•	↑	↑	↑	↑

↑ tested & set if true, cleared otherwise

• not affected

(D3 to D4), or a borrow from the high nibble to the low nibble (D4 to D3) of an 8 bit number. This is used by instructions performing BCD (binary coded decimal) arithmetic.

Bit D2 in the condition code register is the **Zero Flag (ZF)**. This flag is set if the result of the instruction is zero. In the preceding example, the zero flag would have been reset (0) since neither instruction produced a zero result (48 and 89).

The last flag covered now is the **Sign Flag (SF)**, bit D3. This flag indicates the polarity of a signed 7 bit number. With signed 7 bit numbers, the most significant bit (D7) indicates whether the number is positive (D7=0) or negative (D7=1). The sign flag is effectively a copy of D7. In the first add example, the number 48 (01001000) would be considered positive.

It is important to note that when using unsigned 8 bit binary numbers for arithmetic, the sign flag cannot be used to indicate a negative number because the SF is set according to D7. For example, subtracting the number 40 from C9 would cause the sign flag to be set. The binary 1 in D7 of the result would be in the SF, indicating a negative result.

11001001	C9
-01000000	40
10001001	89

A fifth addressing mode, **Implied (IMP)**, has been added to Table 3-1. Instructions that don't use an operand located in memory are under implied addressing. The instruction ABA, for example, adds the contents of ACCA and ACCB and places the result in ACCA. The operands are specified (implied) by the instruction itself.

The subtraction instructions are in Table 3-2. These are very similar to the add instructions in that an operand is subtracted from one of the accumulators and the result is returned to that accumulator. When a larger number is subtracted from a smaller number, the carry flag is set to indicate that a borrow has occurred. The result is in 2s complement form. The CF can therefore be used as an indication of whether or not an unsigned 8 bit result is positive or negative, that is, if a carry (borrow) has occurred, the result is negative.

Subtraction can be used to determine the relation between 2 numbers. For example, the zero flag, if set after a subtraction, would indicate the numbers were equal. If the CF (borrow) flag was set, then the source operand was larger than the destination operand. If the CF and the ZF were both zero, then the source operand was larger than the destination operand. It is through the use of the instructions that use the flags (called *conditional*



**TABLE 3-2**  
**Subtraction Instructions**

OPERATION	MNEM	ADDRESSING MODES					Cond code res					
		IMM	DIR	IND	EXT	IMP	5	4	3	2	1	0
		op ~ =	op ~ =	op ~ =	op ~ =	op ~ =	H	I	N	Z	V	C
SUBTRACT	SUB A	80 2 2	90 3 2	A0 5 2	B0 4 3		•	•	↑	↑	↑	↑
	SUB B	C0 2 2	D0 3 2	E0 5 2	F0 4 3		•	•	↑	↑	↑	↑
Subtractor Acmltrs	SBA					10 2 1	•	•	↑	↑	↑	↑

branching) that programs make decisions. These instructions are covered in Lab 4.

Because so many factors can be determined by subtraction, there is a special group of instructions that subtracts two operands and sets the flags, but discards the result. The 2 operands remain unchanged. They are called **Compare (CMP)** instructions and are shown in Table 3-3.

**TABLE 3-3**  
**Compare Instructions**

OPERATION	MNEM	ADDRESSING MODES					Cond code res					
		IMM	DIR	IND	EXT	IMP	5	4	3	2	1	0
		op ~ =	op ~ =	op ~ =	op ~ =	op ~ =	H	I	N	Z	V	C
COMPARE	CMP A	81 2 2	91 3 2	A1 5 2	B1 4 3		•	•	↑	↑	↑	↑
	CMP B	C1 2 2	D1 3 2	E1 5 2	F1 4 3		•	•	↑	↑	↑	↑
Compare Acmltrs	CBA					11 2 1	•	•	↑	↑	↑	↑

The last group of arithmetic instructions covered now are in Table 3-4. The **Increment (INC)** instructions add 1 to the operand. The operand can be in an accumulator or in memory. The **Decrement (DEC)** instructions subtract 1 from the operand.

**TABLE 3-4**  
**Increase and Decrease Instructions**

OPERATION	MNEM	ADDRESSING MODES					Cond code res					
		IMM	DIR	IND	EXT	IMP	5	4	3	2	1	0
		op ~ =	op ~ =	op ~ =	op ~ =	op ~ =	H	I	N	Z	V	C
INCREMENT	INC			6C 7 2	7C 6 3		•	•	↑	↑	*	•
	INC A					4C 2 1	•	•	↑	↑	*	•
	INC B					5C 2 1	•	•	↑	↑	*	•
DECREMENT	DEC			6A 7 2	7A 6 3		•	•	↑	↑	*	•
	DEC A					4A 2 1	•	•	↑	↑	*	•
	DEC B					5A 2 1	•	•	↑	↑	*	•

\* Condition Code Notes—See instruction set in Appendix A for explanation

It is important to realize that the INC and DEC instructions *do not* affect the carry flag. The carry flag remains as it was prior to the instruction.

Also note what happens when the operand is FF before being incremented. The result is 00, with the ZF indicating a zero result. When the operand is 00 before a decrement, the result is FF the ZF is reset (not zero).

---

## LAB PROCEDURE

1. Enter the following numbers into the specified memory locations.

address	object code
0020	05
0021	00
0022	06
0023	CF
0024	FF

2. Enter the following machine code, then single step through the instructions, recording the results as indicated.

### Program 3-1:

address	object code	mnemonic
0000	CE	LDX,0020 (IMM)
0001	02	High data byte
0002	00	Low data byte
0003	A6	LDA A,00 (IND)
0004	00	Offset
0005	8B	ADD A,30 (IMM)
0006	30	Data byte
0007	A7	STA,01 (IND)
0008	01	01 offset
0009	AB	ADD A,00 (IND)
000A	00	Offset
000B	AB	ADD A,02 (IND)
000C	02	Offset
000D	BB	ADD A,0023 (EXT)
000E	20	High address byte
000F	37	Low address byte
0010	4C	INC A (IMP)
0011	6C	INC 04 (IND)
0012	01	Offset

### Instruction 1:

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

### Instruction 2:

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 4:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 5:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 6:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 7:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 8:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 9:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

3. Enter the following numbers into the specified memory locations.

address	object code
0020	C8
0021	41
0022	87

4. Enter the following machine code, then single step through the instructions recording the results as indicated.

**Program 3-2:**

address	object code	mnemonic
0000	CE	LDX 0020 (IMM)
0001	02	High data byte

address	object code	mnemonic
0002	00	Low data byte
0003	A6	LDA A 00 (IND)
0004	00	Offset
0005	A0	SUB A 01 (IND)
0006	01	Offset
0007	B0	SUB A 0022 (EXT)
0008	02	High address byte
0009	02	Low address byte
000A	4A	DEC A (IMP)
000B	4C	INC A (IMP)
000C	AB	ADD A 02 (IND)
000D	00	Offset
000E	81	CMP A 87 (IMM)
000F	C8	Data byte
0010	A1	CMP A 00 (IND)
0011	00	Offset
0012	A1	CMP A 01 (IND)
0013	01	Offset
0014	B7	STA A 0023 (EXT)
0015	02	High address byte
0016	03	Low address byte

**Instruction 1:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 2:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 4:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 5:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 6:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 7:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 8:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 9:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 10:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_

**Instruction 11:**

ACCA \_\_\_\_\_

Flags:carry \_\_\_\_\_ Auxiliary carry \_\_\_\_\_ sign \_\_\_\_\_

zero \_\_\_\_\_





## LAB 3 ARITHMETIC INSTRUCTIONS

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

*(Questions 1–4 apply to Program 3-1)*

1. How did the first instruction affect the second instruction?
2. Did the fourth instruction store the ASCII code for the operand used in the second instruction? Where was it stored?
3. Explain the state of the condition codes for Instructions 6–9.
4. What operand (number) was used in Instruction 9? Where was the result placed?

*(Questions 5–9 apply to Program 3-2)*

5. Why did the sign flag indicate a negative result when the third instruction was executed?
6. Did Instructions 5 and 6 produce the results expected? Why or why not?
7. Did Instructions 8, 9, and 10 change the operand?
8. What did the flags indicate about the relationship between the operands used in Instruction 8? Instruction 9? Instruction 10?

9. What other addressing modes could be used in Instruction 11 to produce the same results?
10. Which of the flags were affected by the LOAD and STORE instructions in Program 3-1 and 3-2?

---

## EXERCISE

---

### *Program 3-3: Adding 2 Numbers*

1. Draw a flowchart that will allow the addition of any 2 numbers previously stored in memory.
  - a. The 2 numbers must be in memory locations 0070 and 0071.
  - b. The result of the addition must then be stored in memory location 0072.

*Example:* 0070 contains data number 23 and 0071 contains data number 31. After the program has been executed, memory location 0072 should contain the sum, data number 54.

2. Write a **machine language program** from your flowchart.
  - a. Begin the program at memory location 0000.
  - b. Load and run your program to ensure proper operation.

### *Program 3-4: Adding 5 Different Numbers*

1. Draw a flowchart that will allow the addition of 5 different numbers previously stored in 5 alternate memory locations.
  - a. The first number must be stored in memory location 0060, the second number in 0062, the next in 0064, and so on.
  - b. Load 00 in each location between the 5 numbers.
  - c. Store the result in memory location 0069.
2. Write a machine language program from your flowchart.
  - a. Begin your program at memory location 0000.
  - b. Load and run your program to ensure proper operation.

3. *Do not* remove this program from RAM until after the completion of Program 3-5.

### *Program 3-5: Subtracting 4 Numbers*

1. Draw a flow chart that will do the reverse of Program 3-4.
  - a. Using the result of Program 3-4 stored in memory location 0079, subtract each of the next 4 numbers in reverse order.
  - b. Store the results in memory location 0071.
2. Write a machine language program from your flowchart.
  - a. Begin this program at memory location 0040. This will allow you to have both Programs 3-4 and 3-5 in memory at the same time. The programs can be run sequentially with Program 3-5 using the numbers left by Program 3-4.
  - b. Load and run your program to ensure proper operation.



# LAB 4

## BRANCHING INSTRUCTIONS

### OUTLINE

Branching Instructions  
Labels

Conditional Branching  
Loops

### KEY TERMS

Branching Instructions  
Conditional Jump  
Jump Instructions

Label  
Loop  
Nested Loops

Relative Addressing  
Unconditional Jump



Upon completion of this lab, you should know:

1. How to alter the instruction sequence of a program.
2. The process a program uses to make a decision.
3. The operation and use of loops.

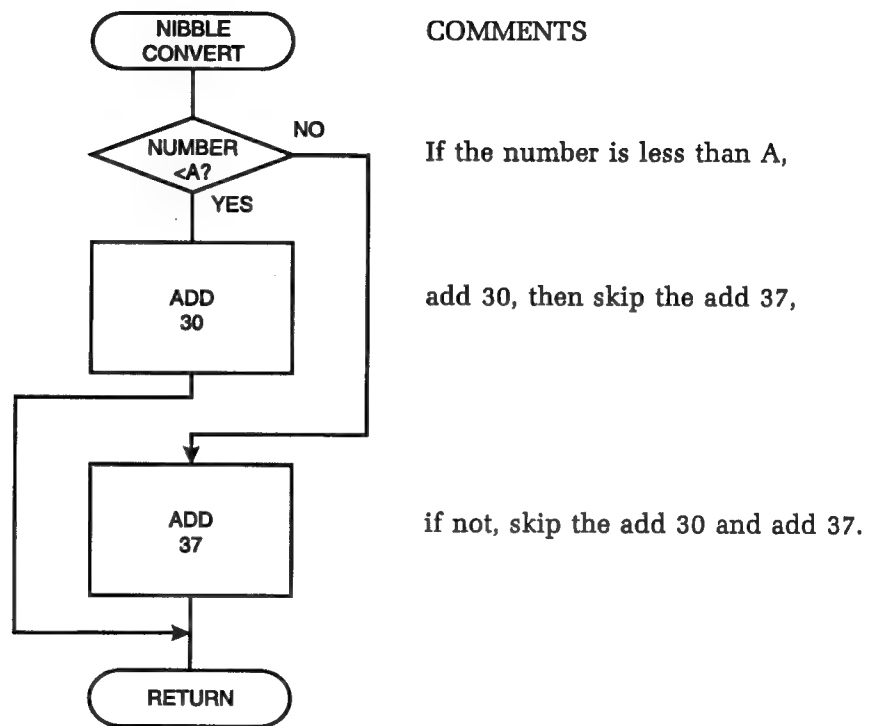
### DISCUSSION

#### Branching Instructions

Examining the flowchart from Lab 3, you will notice several instances where different sequences of instructions need to be executed. In the convert subroutine, the decision of whether to add 30 or 37, based on the value of the hex number (0-9 or A-F), must be made. The sequence that adds the correct conversion factor must then be executed by the CPU. Because all program instructions are sequentially executed, some way of changing the sequence is needed.

The flowchart of the convert subroutine would look more like Figure 4-1 if the way it would be stored in memory is considered.

The instructions that allow us to change the flow of a program are called **jump** and **branching** instructions. There are 2 types of branching instructions: an **unconditional** and a **conditional**. Unconditional jumps always branch to the address contained in the instruction. Conditional jumps branch only if certain conditions of the flags exist. Refer to Appendix A for the mnemonics and op codes for the remaining labs.

**FIGURE 4-1 Convert Subroutine Stored in Memory**

Remember from earlier labs that the program counter contained the address of the next instruction to be executed. If the number in the PC were changed, the next instruction would be fetched from the new address in the PC register. This is what the branching instructions do: change the contents of the PC.

There are 2 methods used to change the address contained in the PC. The first method is called **relative (REL) addressing**. In relative addressing, the instruction includes an offset added to the PC. The offset is the distance to the target address from the branch instructions. The result obtained by adding the offset and the PC replaces the old PC value. The next op code fetch is from the new address contained in the PC.

The offset is an 8 bit number with backward jumps in 2s complement form. When it is added to the PC, the sign (D7) is extended to the upper 8 bits. This produces a branching range of -128d to +127d from the present PC. Because the PC would be incremented after fetching the offset byte, it would contain the address of the instruction *after* the branch instruction.

Address	Op code	Mnemonic
0200	20	BRA (Branch always)
0201	0A	Offset
0202	XX	Next instruction
0203		
.		
.		
.		
020C	XX	Target of branch

In this example, the offset needed to branch from 0200 to 020C would be as follows.

020C	Branch address
<u>-0202</u>	PC after branch instruction
0A	Offset

If a program needs to branch backwards using relative addressing, the offset must be in 2s complement form. Remember that adding a 2s complement number produces the same result as subtraction.

Address	Op code	Mnemonic
024F	XX	Target of branch
0250	XX	
0251		
.		
.		
.		
0258	20	BRA (to 024F)
0259	F5	Offset
025A	XX	Next instruction op code

In this example, the program needed to branch from 025A to 024F.

025A	PC after branch
<u>024F</u>	Branch address
0B	Distance of branch
F5	2s complement

Branches to addresses over 128d locations away must use a JMP instruction. Instead of using relative addressing, as with branch instructions, the jump can use either extended or indexed addressing. In extended addressing, the 16 bit address of the jump location is included with the op code.

Address	Op code	Mnemonic
02F6	7E	JMP 1F00
02F7	1F	High address byte
02F8	00	Low address byte

This instruction would cause the next op code fetch to come from 1F00.

A jump using indexed addressing includes an offset byte with the op code. The offset byte is added to the IX register, then the sum is loaded into the PC. The next fetch will be from the new PC contents, IX plus offset. To use indexed addressing, as always, the IX register must be loaded with the base address first!

Address	Op code	Mnemonic
037F	CE	LDX 1000 (IMM)
0380	10	High data byte
0381	00	Low data byte
0382	6E	JMP A0 (IND)
0383	A0	Offset

This sequence will cause a jump to address 10A0 (IX + offset).

## **LABELS**

When the source code is written using mnemonics, the target address of branching instructions is usually unknown. Instead of inserting numbers to represent the address, most often a **label** is used. A label is a group of characters that represents some aspect of where the jump is going.

Labels used for the nibble convert subroutine are:

```
ADD37
END
ADD30
```

The label *must* be converted to an address when the source code is converted to object code. The use of labels is for convenience when writing the program.

## CONDITIONAL BRANCHING

To determine which add instruction to execute in the HEX to ASCII program, a conditional branch is needed. A conditional branch is a branch that may or may not change the PC, depending on the state of the condition codes at the time the instruction is executed.

If the condition tested is true, the branch occurs. If the condition is false, the instruction after the branch is executed next.

All conditional branches use relative addressing. That limits conditional branches to +127d to -128d.

Analyzing the decision the program must make provides several ways to make the decision.

*Method 1:* The program can check to see if the number is an A, if so, add 37; then it would check for B, if so, add 37; and so on until it checked for F, if so, add 37; if it did not equal any of these, then the number must be 0-9, so add 30.

In this method, each decision block would consist of a compare with the number A, then B, and so on, followed by a BEQ (branch if the result of the compare was zero) to the address of the add 37 instruction.

```

CMP  A,0A (IMM)
BEQ  ADD37
CMP  A,0B (IMM)
BEQ  ADD37
.
.
.
.
CMP  A,0F (IMM)
JZ   ADD37
ADD  A,30 (IMM)
BRA  END
ADD37: ADD  A,37
END:
```

However, this method is not very efficient, because of the number of instructions needed.

If the decision the program needs to make is analyzed further, it can be seen that the program can look to see how the number being converted relates in value to either 9 or A.

If it is equal to or less than 9, 30 must be added; otherwise, add 37. It can also be said that if the number is greater in value than 9, add 37; otherwise, add 30. Looking at it from the standpoint of A, if the number is equal to or greater than A, add 37; otherwise, add 30. If the number is less than A, add 30; otherwise, add 37.

*Method 2:* If it is equal to or less than 9, add 30; otherwise, add 37.

```

CMP  A,09 (IMM) ; Subtract 09 from ACCA
BEQ  ADD30      ; If result is zero (#=9) add 30
BCS  ADD30      ; If result produces a borrow (#<9) add 30
ADD  A,37 (IMM) ; Else add 37
BRA  END        ; Jump around add 30
ADD30:
ADD  A,30 (IMM) ; Adds 30
END:

```

*Method 3:* If the number is less than A, add 30; otherwise, add 37.

```

CMP  A,0A (IMM) ; Subtract 0A from ACCA
BCS  ADD30      ; If borrow is produced add 30 (#<A)
ADD  A,37 (IMM) ; Else add 37
BRA  END        ; Jump around add 30
ADD30:
ADD  A,30 (IMM)
END:

```

As you can see, there are many ways to make decisions. All of these methods will produce the same result—they just do it differently.

The method that uses the fewest instructions is normally the best way. The key to making efficient programs is to analyze the best way of manipulating the number to affect the needed flags. Quite often the compare instructions will be used, but choosing the number to subtract can make a lot of difference (as in Method 3 vs. 2).

## LOOPS

Another use for conditional branches is with parts of a program that must repeat a certain number of times. The part of the program that repeats itself is called a **loop**. A common use for loops is to set up timing delays. A timing delay could be constructed by taking a core of instructions (selected to produce a desired time base) and executing that core the number of times needed for the desired delay time. If, for example, a time delay of 100 milliseconds is needed, the following approach could be used.

The first step is to use a few instructions to form a time base to be used over and over for the specified delay. Examining the instruction set in Appendix A, it can be seen that the instructions take 2 to 8 bus cycles. Each bus cycle takes 1μsec with a 1MHz clock frequency. Each instruction takes from 2μs to 8μs to execute.

The following sequence of instructions will give a time delay of 1002μs with a microprocessor running at 1.00Mhz.

Memory Address	Contents	Comments
437E	86	LDA A,64 (IMM)
437F	64	Data byte
4380	DE	LDX,20 (DIR)
4381	20	Address byte
4382	4A	DEC A (IMP)
4383	27	BNE (4380) (Rel)
4384	FA	Offset of branch

The LDA A instruction will take 2 bus cycles, the LDX 4 bus cycles, DEC A 2 cycles, and BNE 4 cycles. The ADD B, DEC A, and BNE instructions will repeat until the number loaded into A in the first instruction is decremented to zero, 100 times.

	2	for LDA A,(64H=100d)
+	400	for LDX, executed 100d times
+	200	for DEC A executed 100d times
+	400	for BNE executed 100d times
	1002	total bus cycles
X	1	μ second per bus cycle
	1002	μ seconds total delay

The 2μs can be dropped and use the sequence as a rounded 1000 μs delay (this is 99.8 percent accurate). If this sequence is now repeated 100d times, a total delay of 100 milliseconds will result.

Memory Address	Contents	Comments
437C	C6	LDA B,64 (IMM)
437D	64	Data byte
437E	86	LDA A,64 (IMM)
437F	64	Data byte
4380	DE	LDX 20 (DIR)
4381	20	Address
4382	4A	DEC A (IMP)
4383	27	BNE (4380)(Rel)
4384	FA	Offset of branch
4385	5A	DEC B (IMP)
4386	27	BNE (437E)(Rel)
4387	F6	Offset branch

The inner LOOP (LDA, LDX, DEC A, BNE) is repeated ACC B times, 100d in this case, giving a total time delay of 100 milliseconds. Putting a loop inside of a loop (ACCA loop inside of the ACCB loop) is called **nesting**, and these are called **nested loops**.

## LAB PROCEDURE

- Convert the following source code into object code, then enter it into memory. This will be the source code for the nibble convert subroutine in the HEX to ASCII program.

### Program 4-1:

```

CMP A,0A (IMM) ; Subtract 0A from ACCA
BCS ADD30      ; If borrow is produced add 30 (#<A)
ADD A,37 (IMM) ; Else add 37
BRA END        ; Jump around add 30
ADD30:
ADD A,30 (IMM)
END:

```

- Put the number 07 into the ACCA register, then single step through the program, recording the results.

### Instruction 1:

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

**Instruction 2:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

3. Repeat the process with 0C in the AL register.

**Instruction 1:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

**Instruction 2:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

**Instruction 4:**

ACCA \_\_\_\_\_ carry flag \_\_\_\_\_

4. Enter the following program.

**Program 4-2:**

Memory Address	Contents	Comments
0000	C6	LDA B,64 (IMM)
0001	64	Data byte
0002	86	LDA A,64 (IMM)
0003	65	Data byte
0004	DE	LDX 20 (DIR)
0005	20	Address
0006	4A	DEC A (IMP)
0007	27	BNE (0004)(Rel)
0008	FA	Offset of branch
0009	5A	DEC B (IMP)
000A	27	BNE (0002)(Rel)
000B	F6	Offset branch

5. Run the program at full speed, timing how long it takes to run. Don't forget to include a breakpoint at 000C.
6. Record the value of the following registers.
- ACCA \_\_\_\_\_ ACCB \_\_\_\_\_ zero flag \_\_\_\_\_
7. Adjust the value loaded into ACCA to produce a 1 second delay.
-





## LAB 4 BRANCHING INSTRUCTIONS

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

*(Questions 1–3 refer to Program 4-1)*

1. Why did the BCS branch occur when ACCA contained 07?
2. Why did the BCS branch not occur when ACCA contained 0C?
3. How is the PC changed with each of the branching instructions?

*(Questions 4–6 refer to Program 4-2)*

4. What was the exit condition for each loop?  
inner \_\_\_\_\_  
outer \_\_\_\_\_
5. How many times was the DEC A instruction executed?
6. Explain the relationship between the accumulator values needed to produce a 1 second delay and the trainer's clock frequency.

### EXERCISE

---

1. Multiplication can be accomplished by repeated addition. Using the process of repeated addition, construct a program to multiply the number in ACCA by the number in ACCB.



## LAB

# 5

# SUBROUTINES

## OUTLINE

Subroutines  
Stack

Keyboard Subroutines  
Display Subroutines

## KEY TERMS

Branch to Subroutine (BSR)  
Call  
First In Last Out (FILO)  
Jump to Subroutine (JSR)

Pull REG (PUL)  
Push REG (PSH)  
Return

Return from Subroutine (RTS)  
Stack  
Stack Pointer



Upon completion of this lab, you should know:

1. How branch to subroutine (BSR) and jump to subroutine (JSR) instructions are used.
2. How the return from subroutine instruction is used.
3. The use and operation of the stack.

## DISCUSSION

### Subroutines

The next branching instructions are normally used with each other, the **branch** or **jump to subroutine** and the **return from subroutine**. Looking at the HEX to ASCII flowchart again, it can be seen that the convert subroutine must be used 2 times, once for each nibble. The instructions used to change the PC to the beginning address of the subroutine are the **branch to subroutine (BSR)** or **jump to subroutine (JSR)**. Branching to a subroutine is often referred to as a **call** to the subroutine. The BSR uses relative addressing, while the JSR can use either indexed or extended addressing. The BSR and JSR do an additional function compared to the branches covered in Lab 4—save the address of the instruction after the branch instruction. They do this so the program can return to the instruction after the branch when the subroutine is finished. When the **return from subroutine (RTS)** is executed, the return address, saved by the branch, will be restored back into the PC.

Memory address	Contents	Comments
0133	BD	Op code for JSR (EXT)
0134	02	High byte of convert subroutine address
0135	00	Low byte of convert subroutine address
0136	XX	Op code for next instruction
.	.	.
.	.	.
0200	XX	Beginning of convert
.	XX	.
.	.	.
.	.	.
.	.	.
.	39	Op code for RTS

The action that would take place with this sequence of instructions is as follows. When the op code for the JSR is decoded, the PC is incremented to the high byte of the address (PC=0134), which is read into a temporary register in the CPU. PC is incremented for the low byte (0135), then PC is incremented again so it points to the next instruction (0136). But instead of fetching the instruction as normal, the CPU stores the old PC in R/W memory. The address included with the JSR instruction (0200) is then loaded into the PC and a fetch is executed from the new address (0200).

When the RTS instruction is executed at the end of the convert subroutine, the address that was in PC (0136) and saved in memory is returned from memory to the PC. The next instruction fetched will now be the one after the JSR (0136).

With the JSR/RTS pair, a subroutine can be called from anywhere in the program and as long as there is an RTS at the end of the subroutine, the program sequence returns to the instruction after the call.

## STACK

The register used to determine where in memory the return address will be stored is called the **stack pointer**. The area of memory used to store the return address is called the **stack**.

The order of how the numbers are stored on the stack is the reverse of how a program is stored. The instructions in a program progress from lower to higher memory address, while sequential stack writes move down in memory (to lower addresses).

Before examining the details of stack operations, there are 2 data transfer instructions that use the stack, the **push reg** (PSH) and **pull reg** (PUL) instructions. A push instruction will write the specified accumulator onto the stack, and a pull will read a number from the stack and move it to the designated accumulator.

The JSR/BSR and push instructions are similar to each other in how the stack operation occurs, as are the RTS and pull instructions. When a JSR or PSH instruction is executed, the low byte of the PC or the content of the designated accumulator is written into the memory location addressed by the number contained in the stack pointer. The high byte of the PC is then written into the location, which is 1 number below the stack pointer address. The stack pointer is decremented twice for a JSR and once for a push.

If, for example, the stack pointer (SP) has the number FF2C in it, and the instruction JSR is executed, the following action will occur.

The low byte of PC will be written to address FF2C.

The SP will be decremented to FF2B.

The high byte of PC will be written to address FF2B.

The SP will be decremented to FF2A.

SP before the JSR is FF2C.

Address	Contents
FF2C	PCL
FF2B	PCH
FF2A	XX

SP after the JSR is FF2A.

The SP will now be pointing to the next entry address on the stack (FF2A). The order in which the bytes are stored in memory is just like all 2 byte numbers, that is, the high byte will be stored at the lower memory address.

The pull and RTS do just the opposite of the push and JSR. Because the SP is pointing to the next entry on the stack, the SP is incremented to the address where the last write occurred. A read will occur, placing the result in PC H. The SP is incremented again, followed by a read which places the result in PC L.

The terminology used with the stack may seem backwards, but the “top” of the stack is actually the last entry, which is in the lowest memory address. The SP, therefore, always points to the top of the stack (the address of the next entry onto the stack).

The push instructions are used to save the contents of the accumulators mainly when subroutines are called. Many subroutines use both registers to perform their function, so the numbers in the used registers will be lost if they are not saved somehow. To do this, one or more push instructions could be executed before a JSR or BSR.

```
PSH  A
PSH  B
JSR  (subroutine)
```

To restore the contents of the registers after the JSR, pull instructions would be used.

```
PSH  A
PSH  B
JSR
PUL  B
PUL  A
```

Notice the order of the pushes as compared to the pulls—they are in reverse order. If the previous sequence of instructions were executed, it would result in the following action.

	Memory	Contents	Instruction
SP before PSH A	FF2C	ACCA –	PSH A
SP after PSH A, before PSH B	FF2B	ACCB –	PSH B
SP after PSH B, before JSR	FF2A	PCL –	JSR
	FF29	PCH –	JSR
SP after JSR	FF28		

The SP points to FF2C, before PSH A is executed. When PSH A is executed, the contents of ACCA are written to FF2C, the present address in SP. After writing to FF2C, the SP is decremented to FF2B. The PSH B instruction will write ACCB in FF2B and decrement the SP to FF2A. The JSR instruction will cause the CPU to write PCL to FF2A and PCH to FF29. The SP will be decremented to FF28. The PC value written will be the address of the PUL B op code.

The RTS at the end of the subroutine would cause the CPU to increment SP from FF28 to FF29. A memory read from FF29 will place the high return address byte of the PUL B instruction into PCL. The SP is then incremented to FF2A. A read from FF2A will return the low address byte to PCL.

The PUL B instruction will cause the CPU to increment SP to FF2B, then load ACCB with the contents of the memory address. Finally, the PUL A instruction will increment SP to FF2C and load ACCA with the number stored at that address. Now the accumulators will have the same numbers in them as they did before the JSR.

This way of putting numbers in and out of a storage area is called **first in last out (FILO)**. That is, the first number stored will be the last one retrieved if the storage locations are accessed sequentially.

Although this approach will work, it is generally considered more efficient and better form to do the pushes and pulls inside the subroutine. The registers used in the subroutine would be pushed at the beginning of the subroutine and pulled in reverse order before the RTS instruction. In this manner, only the registers used in the subroutine need be pushed and pulled.

---

## KEYBOARD SUBROUTINES

Most trainers provide subroutines with the monitor program, which simplifies the use of the keyboard for inputting characters. The subroutine is "called" with a JSR instruction that specifies the subroutines address. When the subroutine returns to the calling program, one of the accumulators (usually ACCA) will contain the encoded value of the key. The main program can then use the result of the keyboard subroutine for various purposes.

---

## DISPLAY SUBROUTINES

Many trainers also include display subroutines with the monitor program. The data to be displayed is stored in memory prior to calling the display subroutine. The location of the display data is "pointed to" by an internal register, such as IX, or it is stored in specific memory locations determined by the display subroutine. As with other trainer-specific functions, the exact use of keyboard and display subroutines will vary between trainers.

---

## LAB PROCEDURE

1. Load the following program into RAM starting at address 0000.

### Program 5-1

#### Listing

```
LDA A,11 (IMM)
LDA B,22 (IMM)
```

**Listing**

```

PSH  A
PSH  B
LDA  A,00 (IMM)
LDA  B,00 (IMM)
PUL  B
PUL  A

```

- Use the R command to set the Stack Pointer (SP) to 007F.
- Single step through the program and record the contents of the requested registers and memory locations *after* executing each second step.

STEP #2   ACCA \_\_\_\_\_   ACCB \_\_\_\_\_   SP \_\_\_\_\_  
               007F \_\_\_\_\_   007E \_\_\_\_\_

STEP #4   ACCA \_\_\_\_\_   ACCB \_\_\_\_\_   SP \_\_\_\_\_  
               007F \_\_\_\_\_   007E \_\_\_\_\_

STEP #6   ACCA \_\_\_\_\_   ACCB \_\_\_\_\_   SP \_\_\_\_\_  
               007F \_\_\_\_\_   007E \_\_\_\_\_

STEP #8   ACCA \_\_\_\_\_   ACCB \_\_\_\_\_   SP \_\_\_\_\_  
               007F \_\_\_\_\_   007E \_\_\_\_\_

**Program 5-2**

- Load the following program into RAM. Set the SP to 005F before single stepping through the program.

**Listing**

```

LDA  A,0A (IMM)
ADD  A,25 (IMM)
PSH  A
PUL  B

```

- Single step through the program and record the contents of each indicated register and memory address after each instruction has been executed.

	Registers		Memory	Flags
	ACCA	ACCB	005F	
#1	_____	_____	_____	_____
#2	_____	_____	_____	_____
#3	_____	_____	_____	_____
#4	_____	_____	_____	_____

- Practice using the display and keyboard routine of the trainer in use.
-





## LAB 5 SUBROUTINES

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

1. Define RAM stack and stack pointer.
2. What size register and how many memory locations are always involved with a push operation, and why?  
  
Registers  
  
Memory locations
3. When several push instructions are executed, which push will write to the lowest address (first or last)? Explain why.
4. Why must you pull the stack in the reverse order that you pushed it?

### EXERCISE

---

1. Construct a program that will accept 4 characters from the keyboard. As each character is received, it should be stored in sequential memory locations. After the fourth character is input, display all 4 characters on the trainer's display.



# LAB 6

## LOGIC, SHIFT, AND ROTATE INSTRUCTIONS

### OUTLINE

Logic Instructions

Rotate and Shift Instructions

### KEY TERMS

AND  
Arithmetic Shift Left (ASL)  
Arithmetic Shift Right (ASR)  
Exclusive OR (EOR)

Logic Instructions  
Logic Shift Right (LSR)  
OR  
Masking

Memory Operand  
Rotate  
Shift



Upon completion of this lab, you should:

1. Be able to predict the result of logic instructions.
2. Understand masking.
3. Understand the use and result of shift instructions.
4. Understand the use and result of rotate instructions.

### DISCUSSION

#### Logic Instructions

This section will present more of the 6800 instruction set. Many of these instructions are used with I/O operations, but are also used for many other functions.

The first group covered is **logic instructions**. The format is very similar to the addition and subtraction instructions. The logic instructions will **AND**, **OR**, or **exclusive OR (EOR)** a **memory operand** with one of the accumulators. The specified accumulator will contain the result of the operation. The memory contents will remain unchanged. The same addressing modes used for ADD and SUB instructions can also be used for logic instructions.

The logic instructions operate on the individual bits of the numbers. The result of 1 bit position will have no effect on the other bit positions. To understand how these instructions function, we can use the example of

ANDing 2 numbers, 72 and 9A, as shown in Table 6-1. Table 6-2 summarizes the result of logic operations.

TABLE 6-1

		D7	D0
	72	01110010	
AND	9A	<u>10011010</u>	
result	12	00010010	

TABLE 6-2  
Logic Instructions Bit Combinations

ACCX	MEMORY	Result in destination		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

An important use of the AND instruction is in a process called **masking**. Quite often a program needs to perform operations on fewer than 8 bits. Since the instructions normally specify an 8 bit register, the number must be modified before the required operation.

In the HEX to ASCII program, for example, the two hex numbers must be in the form of 0000XXXX. The lower nibble (XXXX) would then have 30 or 37 added to it. The AND instruction can be used to mask out the high nibble. If ACCA contained the number C9 and the 9 is to be converted, the C can be replaced with zeros by ANDing ACCA with 0F.

ACCA	C9	11001001
immediate number	0F	<u>00001111</u>
result		00001001

The program can now add a 30 to ACCA to produce the ASCII result of 39.

When an individual bit needs to be tested, the other bits can be masked out with an AND instruction. The zero flag can then be tested with a conditional jump to see if the bit was a 1 or 0. If ACCA contained the data, and bit 3 (D2) needed to be tested, the following instructions could be used.

```
AND  A,04 (IMM)
BNE  routine
```

All the bits except D2 would be zero after the AND.

ACCA	XXXXXXXX
AND 04	<u>00000100</u>
result	00000X00

If D2 was a 1, the jump would occur as the result would be 04. If D2 was a 0, the jump would not occur as the result would be 00. If all 8 bits needed to be tested, 8 different and/jump combinations would be required: 01 for D0, 02 for D1, and so on to 80 for D7.

## ROTATE AND SHIFT INSTRUCTIONS

The 6800 also has instructions which move bits to the right or left. These are called **shift** and **rotate** instructions. Figure 6-1 illustrates this group of instructions. The operand can be a register or memory location. Only the specified operand and flags are changed.

**Arithmetic Shift Left (ASL)**, and **Arithmetic Shift Right (ASR)** are used with signed binary numbers. With ASL, the sign bit (D7) is shifted into the carry flag (CF). A zero is shifted into D0. The ASR instruction shifts D7 into D7 and D6. The sign bit (D7) remains unchanged, and D6 is a copy of it. The low bit, D0, is shifted into the carry flag.

**Logic Shift Right (LSR)** is very similar to the arithmetic shift right. The difference between ASR and LSR is what is shifted into D7. The arithmetic shift ASR keeps D7 the same, while the logical shift LSR puts a zero into D7. An example of the shifts is illustrated following. The ACCB register has 10100101 in it *before* each shift, and the CF has a zero.

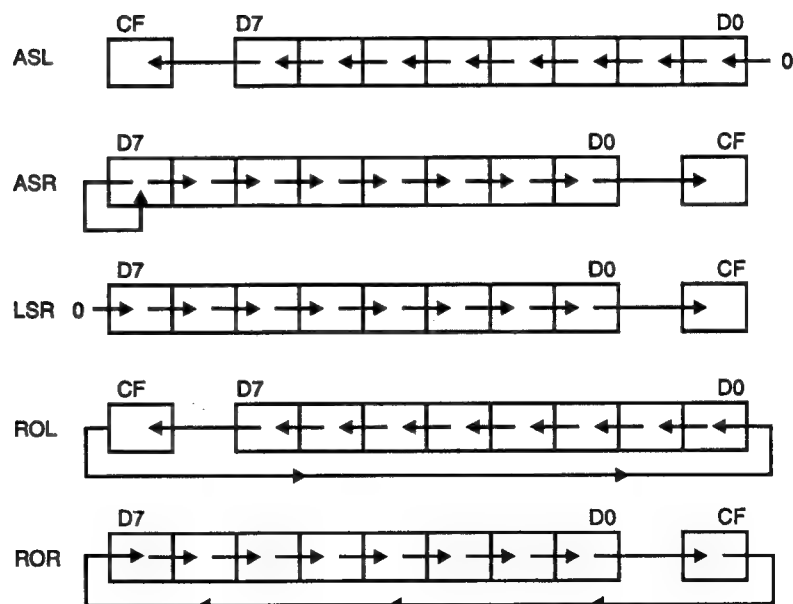
ACCB before shift	10100101	CF=0
ACCB after ASL B	01001010	CF=1 (from D7)
ACCB after ASR B	11010010	CF=1 (from D0)
ACCB after LSR B	01010010	CF=1 (from D0)

With the shift instructions, once a bit is shifted out of the carry flag, it is lost. The bit most distant from the carry flag is replaced with a zero or sign bit. The rotate instructions are like the shift instructions in that bits are shifted left or right. But instead of replacing a bit with a zero (or sign), the bits are rotated in a circular manner.

The ROR and ROL instructions rotate the bit out of the register (D0 for ROR, and D7 for ROL) and into the CF. The CF is rotated into the opposite bit (D7 for ROR, and D0 for ROL). After 9 rotates, the number in the register is in its original form. The CF is treated as a ninth bit.

There are many miscellaneous uses for rotate and shift instructions. The shifts can be used for multiplying and dividing. Each shift to the left is multiplying by 2, each shift to the right divides by 2. A combination of shifts and adds can be used to multiply by other factors. For example, the

**FIGURE 6-1** Shift and Rotate Examples



following sequence of instructions multiplies the number in memory location F0 by 10.

```
LDA B,F0 (Dir)      ; Get number
ASL B               ; Multiply ACCB by 2
ASL B               ; Multiply ACCB by 2 again (2×2=4)
ADD B,F0 (Dir)      ; Add number to ACCB (2×2)+1=5)
ASL B               ; Multiply ACCB ×2 ((2×2)+1)×2=10
```

The shifts are also needed in the HEX to ASCII program from Lab 3. The high nibble needed to be in the form 0000XXXX before the conversion factor (30 or 37) could be added.

Four logical shifts to the right will prepare the high nibble in ACCA for conversion.

```
LSR A
LSR A
LSR A
LSR A
```

ACCA would be in the form 0000XXXX after the 4 shifts.

An important use for the rotates is bit testing. Quite often, a single bit is used to indicate a condition. For example, in a burglar alarm system, a single bit can be used to indicate if a door is open (bit=1) or closed (bit=0). If a bit is rotated into the carry flag, a conditional branch (BCC or BCS) can be used to decide whether or not a door is open. The correct sequence of instructions (sound the alarm or check the next door) could then be run.

---

## LAB PROCEDURE

1. Enter Program 6-1.

```
LDA A,A5 (IMM)
AND A,5A (IMM)
LDA A,54 (IMM)
AND A,0F (IMM)
LDA A,54 (IMM)
ORA A,80 (IMM)
LDA A,04 (IMM)
AND A,04 (IMM)
LDA A,00 (IMM)
AND A,04 (IMM)
LDA A,53 (IMM)
EOR A,53 (IMM)
LDA A,53 (IMM)
EOR A,A0 (IMM)
```

2. Single step through the program and record the requested registers and flags before and after each logic instruction is executed.

1st AND

Before    ACCA \_\_\_\_\_    carry flag \_\_\_\_\_  
           zero flag \_\_\_\_\_    sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 2nd AND

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 1st OR

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 3rd AND

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 4th AND

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      AL \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 1st XOR

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 2nd XOR

Before      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

## 3. Enter Program 6-2. Initial numbers are instruction numbers.

```

        LDA A,0A (IMM)
1      ASR A
2      ASL A
3      ASL A
        LDA A,55 (IMM)

```

```

4    ASL A
5    LSR A
    LDA A,02 (IMM)
6    ROR A
7    ROR A
8    ROR A
    LDA A,80 (IMM)
9    ROL A
10   ROL A
11   ROL A
12   ROR A

```

4. Single step through the program and record the requested registers and flags before and after each rotate and shift instruction.

#1

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____
After	ACCA _____	carry flag _____
	zero flag _____	sign flag _____

#2

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____
After	ACCA _____	carry flag _____
	zero flag _____	sign flag _____

#3

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____
After	ACCA _____	carry flag _____
	zero flag _____	sign flag _____

#4

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____
After	ACCA _____	carry flag _____
	zero flag _____	sign flag _____

#5

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____
After	ACCA _____	carry flag _____
	zero flag _____	sign flag _____

#6

Before	ACCA _____	carry flag _____
	zero flag _____	sign flag _____



After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#7

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#8

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#9

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#10

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#11

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

#12

Before     ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

After      ACCA \_\_\_\_\_      carry flag \_\_\_\_\_  
              zero flag \_\_\_\_\_      sign flag \_\_\_\_\_

---



## **LAB 6 LOGIC, SHIFT, AND ROTATE INSTRUCTIONS**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### **QUESTIONS**

---

*(Questions 1–3 refer to Program 6–1)*

1. What is the result of ANDing a hex digit with F? With 0?
2. What is the result of ORing a hex digit with F? With 0?
3. What is the result of EORing a number with itself?

*(Questions 4–5 refer to Program 6–2)*

4. Explain what happens to the carry flag during instructions 1, 2, and 3.
5. Explain what happens to the carry flag during instructions 5 and 6.
6. How many times would an ROR instruction have to repeat to restore the original bit pattern?  
An ROL?
7. What number must be ANDed with a register to test bit D4?

8. What is masking?

9. Shifting a number right 3 places multiplies/divides by what number?

## **EXERCISE**

---

1. Construct a program to perform HEX to ASCII conversions. The program should convert a byte in memory address 0060 into 2 ASCII characters and store them in 0061 and 0062.

# LAB 7

## INPUT/OUTPUT ADDRESSING

### OUTLINE

Bus Cycle  
Device Select Pulse  
Decoding  
Output

Input  
Lab Circuit  
Circuit Operation

### KEY TERMS

Bus Cycle  
Device Select Pulse  
Memory Mapped I/O  
MEMRD

MEMWR  
Polling  
Port

Read/Write (R/W)  
Tri-States  
Valid Memory Address (VMA)



Upon completion of this lab, you should know:

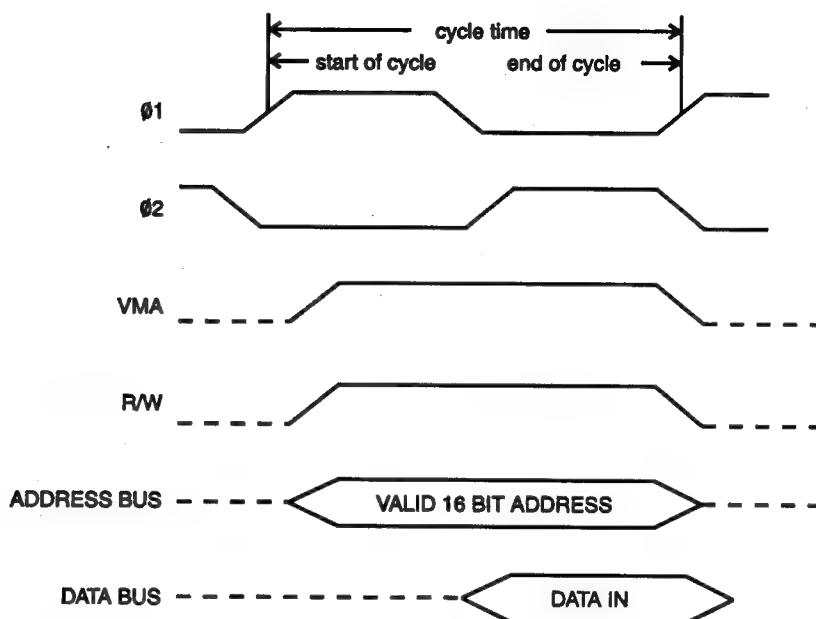
1. The signals used to develop a device select pulse (DSP).
2. Address decoding techniques.
3. How D-latches are used for output.
4. How tri-states are used for input.
5. Timing of I/O bus cycles.

### DISCUSSION

#### Bus Cycle

Interfacing peripheral devices to a microprocessor requires an understanding of the signals used to implement a **bus cycle**. The CPU performs a bus cycle every time it reads from or writes to an external device. External devices include memory circuits as well as input and output circuits. Figure 7-1a shows the signal used in a read bus cycle and Figure 7-1b shows a write bus cycle.

During the time  $\phi 1$  is high, the CPU outputs an address on the address bus. The CPU also outputs a high on the **Valid Memory Address (VMA)** line indicating that the address bus has a valid address on it. The **Read/Write**

**FIGURE 7-1a Input Bus Cycle**

(R/W) control line is put into a high state, indicating a read bus cycle. During the time Ø2 clock is high, the external device should place the data on the data bus. The CPU will latch the number on the data bus into the specified register at the end of the Ø2 clock high time.

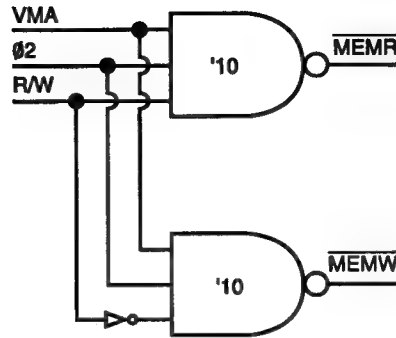
A write bus cycle is identical to a read except for the R/W control line and the direction of data movement. The R/W line goes to a low state during Ø1, indicating a write to the external circuit. During Ø2, the CPU will put the data from the specified register onto the data bus. The external device must latch the number on the data bus into its circuits before Ø2 ends.

Selecting 1 device out of the 64K possible addresses is accomplished by decoding the address bus. The 6800 alone does not distinguish between a memory circuit and an input/output circuit. The CPU treats I/O devices as if they were memory locations. This is called **memory mapped I/O**. The I/O devices are memory addresses *not* used by memory circuits.

To distinguish between I/O devices and memory, the address used for an I/O device is called a **port**. An input device addressed by the number 0080 would be called "input port 0080." There could also be an output port 0080, as the input will be activated by a read control signal, and the output will be activated by a write control signal.

Generating a read and write control signal is accomplished by combining the CPU control signals VMA and R/W with the Ø2 clock. The top NAND gate in Figure 7-2 will generate a **memory read** signal ( $\overline{\text{MEMRD}}$ ) when the CPU is performing a read bus cycle. Inverting the R/W signal generates a **memory write** signal ( $\overline{\text{MEMWR}}$ ) when the CPU is performing a write bus cycle as shown in the bottom NAND gate of Figure 7-2. These 2 signals will be used as control lines for the remaining labs on interfacing.

**FIGURE 7-1b Output Bus Cycle**

**FIGURE 7-2** Circuit to Produce Control Signals

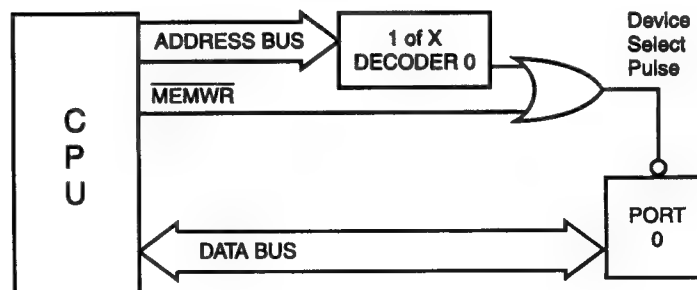
## DEVICE SELECT PULSE

Using the address to select 1 device out of all the possible devices requires the number to be decoded. This decoded number must also be combined with the control signal so that the addressed device will respond at the correct time. The signal that results is used to clock the I/O device to either latch data off the data bus (output) or put data on the data bus (input). This signal is called a **device select pulse (DSP)**. It is a pulse that will last for the duration of the high state of the Ø2 clock.

The circuit in Figure 7-3 is a diagram of a simple output circuit. The decoder is used to decode the address bus and provide a low signal out for the decoded number. In Figure 7-3, the output line 0 is used. This will be low for the address of 0000. Refer to Figure 7-3 as you read the description of the events during each clock phase for this circuit to perform an output to port 0000.

At the beginning of Ø1, the address of 0000 is applied to the address bus. After a propagation delay of a few nanoseconds, the output line 0 would go to a low. This would apply a low to 1 input of the OR gate. The other input is  $\overline{\text{MEMWR}}$ , which is a high during Ø1. This causes the OR gate output to remain in a high state. Referring to Figures 7-1 and 7-2, during Ø1, the VMA line is high and R/W is low. The third input to the control gate, Ø2, is low during Ø1. This is why  $\overline{\text{MEMWR}}$  is still in a low state. When Ø2 goes high, the output of the control gate will go to low. This low input to the OR gate, along with the low from the decoder, will cause the output of the OR gate to go low. This low will now activate the output device to latch the data that is on the data bus. The output of the OR gate is called the device select pulse.

When Ø2 goes to low, the address is removed from the address bus, data removed from data bus, and  $\overline{\text{MEMWR}}$  goes to a high.

**FIGURE 7-3** Basic Output Interface

It is important to understand what signals are used to develop a device select pulse. It is the decoded address (port number) combined with 1 of the control signals ( $\overline{\text{MEMRD}}$  or  $\overline{\text{MEMWR}}$ ). It is used to activate the 1 device connected to it.

The timing of the device select pulse is controlled by the control line,  $\overline{\text{MEMRD}}$  or  $\overline{\text{MEMWR}}$  (which is in phase with  $\phi 2$  clock), gated with the decoded address. The address portion of the device select pulse is the output of a decoder. The decoder is decoding the address, output from the microprocessor, on the address bus.

## DECODING

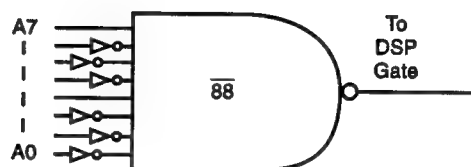
The simplest way of decoding an address is to decode 1 address at a time. The examples in Figures 7-4 through 7-6 show several addresses decoded using this method. This is an easy way to decode, but a system using very many I/O ports would require many ICs to provide the needed decoding.

One way to reduce chip count is to use decoder ICs whenever possible, as shown in Figure 7-7. This requires the ports to be numbered consecutively, as the decoder decodes a sequence of address.

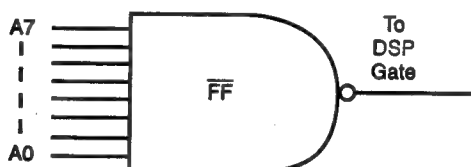
When examining decoder operation, it helps to look at the binary numbers. The NAND gate in Figure 7-7 decodes the upper 6 bits and will have a low out only when the address lines A2 through A7 are all high. The decoder is enabled (low on  $\overline{\text{CS}}$ ) when this condition is met. The decoder will now decode A0 and A1 into 1 of 4 outputs.

	Binary Address	Hex Address
If A2 through A7 are high the decoder is enabled:	111111XX	
The decoder's 0 output is low if the address bus has:	11111100	FC
1 output low if address is:	11111101	FD
2 output low if address is:	11111110	FE
3 output low if address is:	11111111	FF

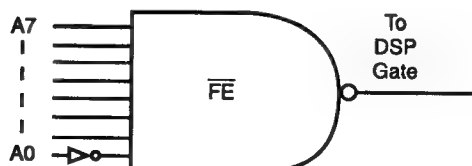
**FIGURE 7-4 Discrete Address Decoding**



**FIGURE 7-5 Discrete Address Decoding**



**FIGURE 7-6 Discrete Address Decoding**

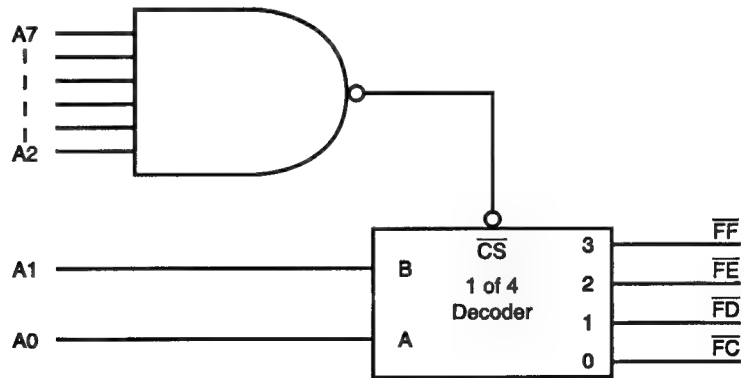




To decode a 16 bit address, a similar method would be used. If, for example, 16 ports were needed and address 0300 through 030F were to be used, the circuit in Figure 7-8 could be used.

The upper NAND gate and inverters would decode the upper part of the address bus for 03. The lower NAND gate would decode A4 through A7 for a 0. The output from these gates would be gated together to be used as an enable signal to the decoder. When the decoder is enabled (030X), it will decode the lower middle for 0–F. One of its outputs would be low; which would depend on the binary pattern on A0 through A3.

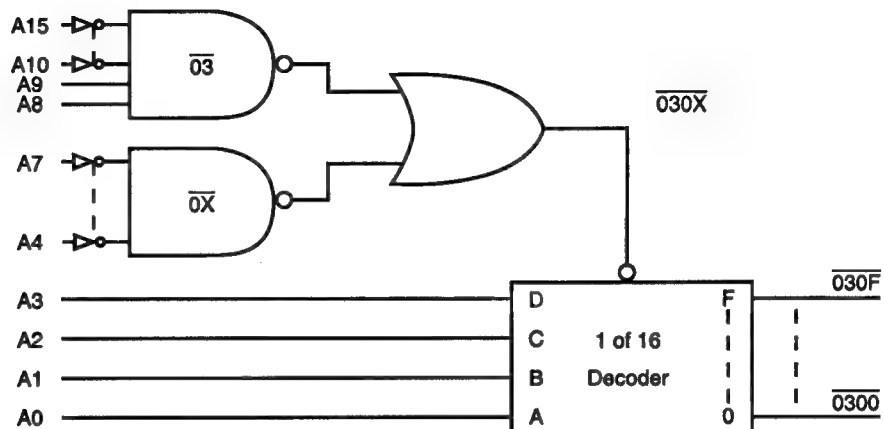
**FIGURE 7-7 Address Decoding Using Decoder IC**



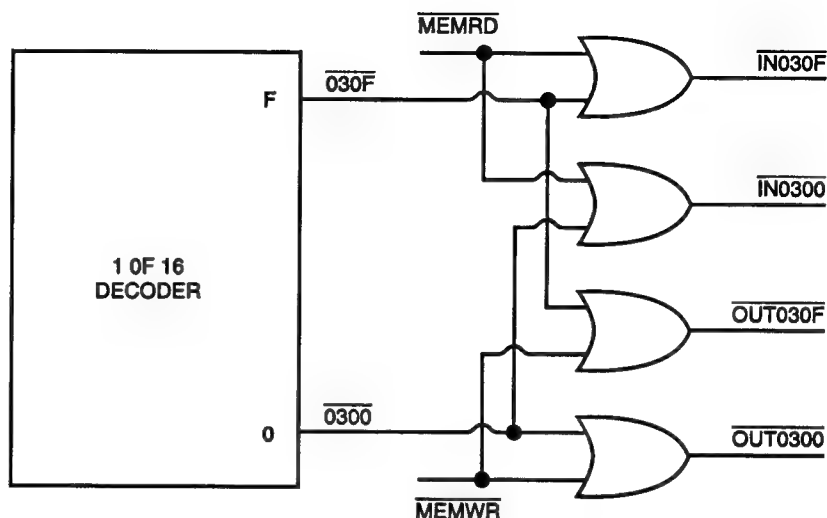
Each of the decoder outputs could be gated with either  $\overline{\text{MEMRD}}$  or  $\overline{\text{MEMWR}}$ . The circuits in the preceding example could be used to generate 32 total device select pulses, 16 for input and 16 for output, (as in Figure 7-9). Note, though, that 32-2 input or gates would be needed.

If, however, 2 decoders were used, 1 for output and 1 for input, instead of gating each output address with a control line, the control line can be used to enable the decoder. See Figure 7-10.

**FIGURE 7-8 Decoding Circuit for 16 Addresses**

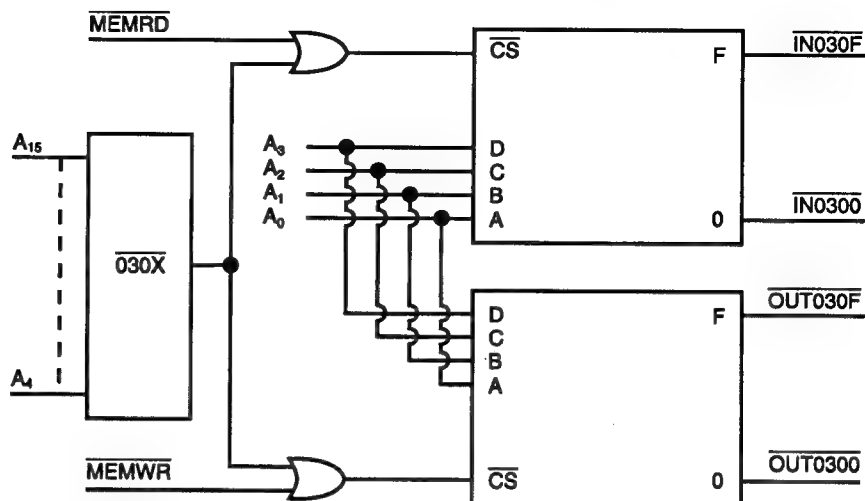


With this circuit, the write decoder is enabled only when the  $\overline{\text{MEMWR}}$  control line goes low. Therefore, the decoder's output is active only when a write bus cycle occurs. The read decoder is activated only when a read bus cycle is executed. In this last circuit, 2 OR gates and 2 decoders performed the same function as 1 decoder and 32 OR gates. Both circuits will accomplish the same results.

**FIGURE 7-9 Decoding Circuit for 16 Input and 16 Output DSP's**

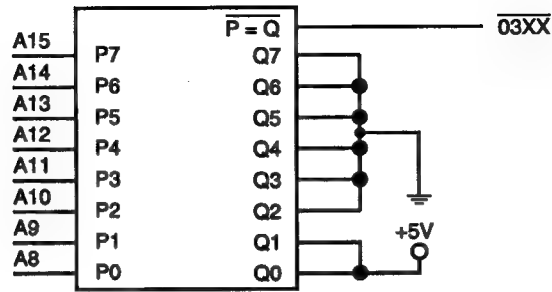
The 8 input AND gates and inverters used to decode A8-A15 could also be replaced with 1 chip, an 8 bit comparator. Whenever a mixture of high and low bits make up the number being decoded, a comparator is preferred. The  $\overline{P=Q}$  signal in Figure 7-11 is the decoded output for the number 03.

There are many ways to generate a device select pulse. All of them use a decoding network to decode the port number on the address bus, and a gating circuit to gate the control line with the decoded address.

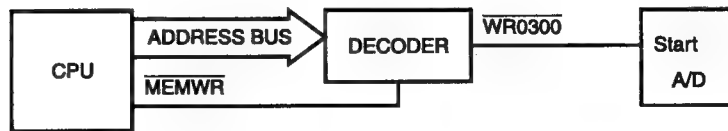
**FIGURE 7-10 Improved 16 Input, 16 Output DSP Decoding Circuit**

## OUTPUT

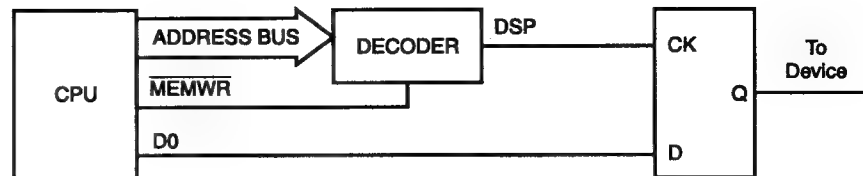
The device select pulse, when used to activate an I/O device, can be used on several different levels. The simplest way is to use the pulse as a signal to a device to start doing something. For example, an A/D converter usually requires a pulse on an input pin to start a new conversion. This can be accomplished with a device select pulse, as shown in Figure 7-12.

**FIGURE 7-11 Address Decoding Using a Comparator**

When a write instruction to port 0300 is executed, the DSP would signal the A/D to start a conversion. In this use of a DSP, no data is transferred, so the data bus is not used. The DSP is used as a signal for an external event to begin or end. This can be anything that requires a simple on/off signal, for example, alarms, motor control, and so on.

**FIGURE 7-12 DSP Used as a Timing Signal**

Another type of on/off control is to use a D-latch to control the status of a device as in Figure 7-13.

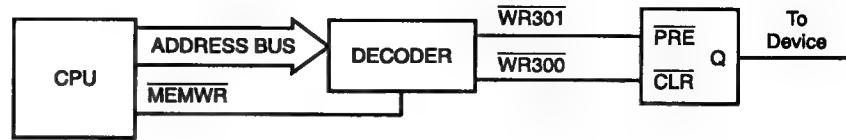
**FIGURE 7-13 Output Latch**

In this circuit, the on or off status is held at a constant level by the D-latch. The data line D0 is used to transfer the status (on/off) from a register in the CPU to the D-latch. With this method, the correct number ( $D0=1$  or  $0$ ) must be in the register before the output occurs. In the previous example, the contents of the registers used in the output were irrelevant as no data was transferred. The method in Figure 7-13 could also be used to put data into a D-latch.

In Figure 7-14, a write to port 0300 would clear the D-latch, while a write to port 0301 would set the D-latch. The disadvantages of this are the need for 2 DSPs, and the limit of control is only 1 device per output. If more devices needed control signals, up to 8 D-latches could be connected to the data bus and all of them be clocked by one DSP. See Figure 7-15.

The programming now becomes more complicated, but less hardware is needed to control more devices.

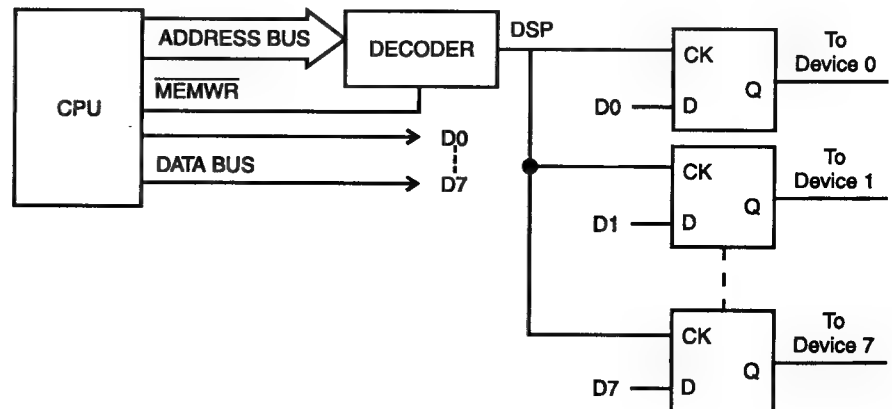
The 8 D-latches used in the preceding example can also be used for outputting a byte from the CPU. The only difference is how the output of the D-latches is used. In the example above, each output of the D-latches controlled the status of a single device, either on or off. The output of the

**FIGURE 7-14 Multiple DSP's Used as Timing Signals**

D-latches can *all* go to a single device for transferring a byte of data. The device can be a printer that receives ASCII data, a D/A converter for controlling the speed of a motor, or a decoding and display circuit to show results of programs, and so on.

In most cases, byte transfers out to ports go to a single IC, which contains 8 D-latches. With this type of IC, there is usually 1 clock input that clocks all latches at the same time. One DSP, therefore, is all that is needed to output a byte to an IC with 8 parallel latches.

If a device select pulse is needed to activate a device without transferring data, a memory write instruction can be used. Since the register contents do not change, nothing happens other than to generate a DSP. The contents of the register are put on the data bus, but since no device connected to the data bus is activated, nothing happens with the data.

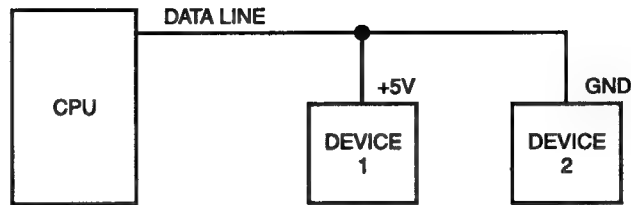
**FIGURE 7-15 8 Bit Output Latch**

## INPUT

When using devices for input, there is a crucial requirement. Only 1 device can be connected to the data bus at a time. If 2 or more signals are applied to the data line at the same time, errors usually occur. If, for example, 2 devices were connected to the data line D0, 1 is a high and the other is a low, the logic level will be something in between. See Figure 7-16.

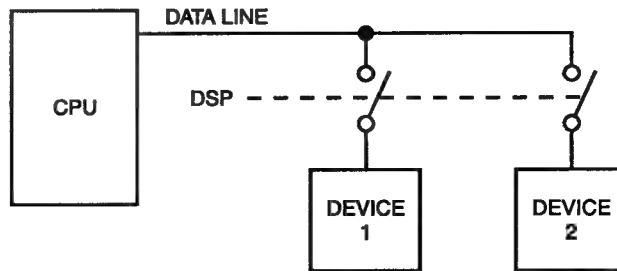
The wire, D0, will have 5V and 0V applied at the same time. In most cases, the actual voltage will be near 0V, or a low logic level. As long as device #2 keeps a low on D0, it will never be able to go to a high state. For this reason, only 1 input device can be connected to the data bus at a time. The simplest approach is to use switches to close or open between the input devices and the data bus. See Figure 7-17.

The DSP would control which switch closed and when. For a byte input, the DSP would control 8 switches (D0–D7) at the same time. Each

**FIGURE 7-16 Bus Contention**

switch would apply either a high or a low to its respective data line. The switch can have 1 of 3 conditions with respect to the data bus. If the switch is open, the input device is disconnected from the data bus. The open switch put an open or high impedance between the input device and the data bus. If the switch is closed, either a high or a low is applied to the data bus, depending on the input device.

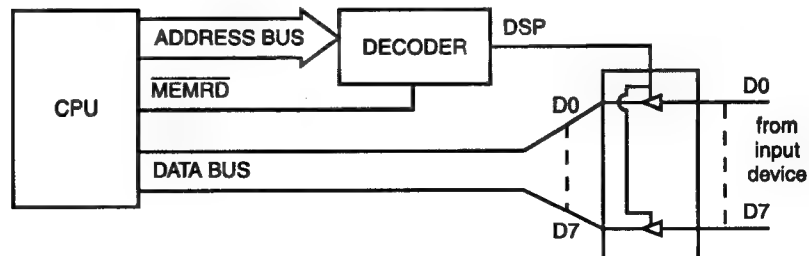
A switch, while used for some input, is much too slow for most input operations. The fastest a mechanical switch can open or close is several milliseconds. Most microprocessors operate in micro-, or even nanoseconds. A mechanical switch used as an input device would be several thousand times slower than the microprocessor. This would cause the microprocessor to wait for the input device to respond to a port read instruction.

**FIGURE 7-17 Switches Eliminate Bus Contention**

There are solid state devices that serve the same function as a switch, but can open and close in nanoseconds. These devices are called *tri-states*. As with switches, there are 3 conditions it can apply to the data bus. If the tri-state is disabled, it is open, or in a high impedance state. If it is enabled, either a high or low logic level is applied to the data bus. The 3 states (open, high, or low) are how the name is derived (*tri-state*).

As with output devices, an input device can be used to input anything between 1 and 8 bits. There are many ICs that have parallel tri-states that are clocked simultaneously. See Figure 7-18.

The symbol for a tri-state is an amplifier (arrowhead) with an enable line connected to it—see Figure 4-18. The enable line controls whether or not it is in a high impedance state. When an input bus cycle is executed, the DSP

**FIGURE 7-18 8 Bit Input Circuit**

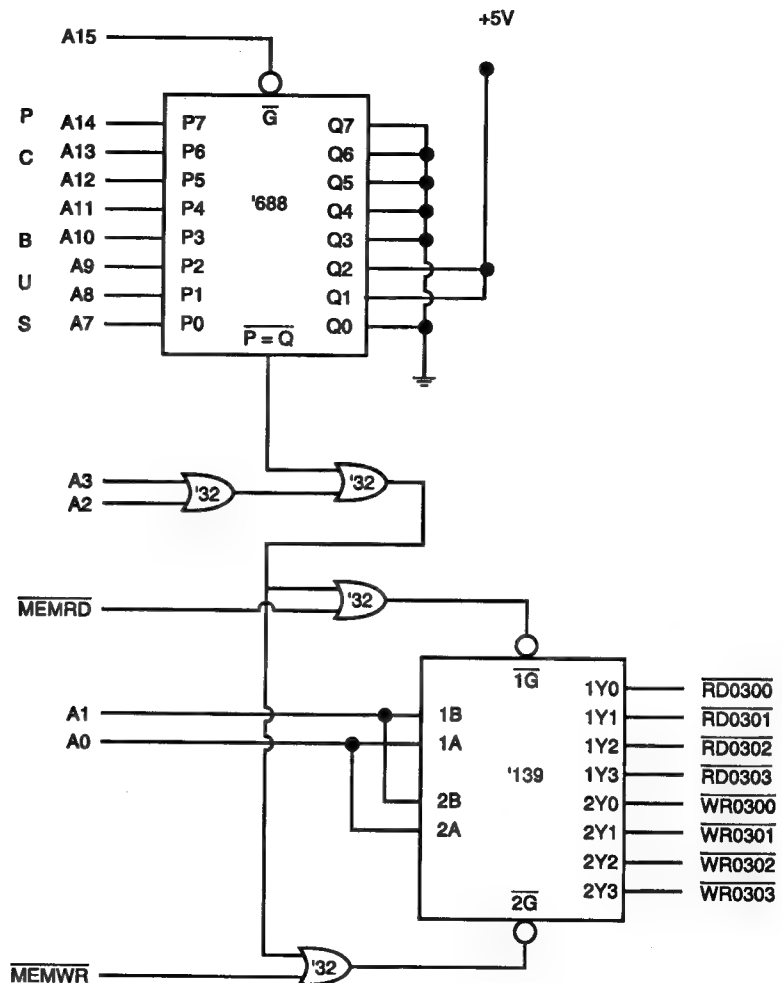
will enable the tri-states. The data on the input of the tri-state (high or low) will be applied to the data bus. This number is latched into the CPU register.

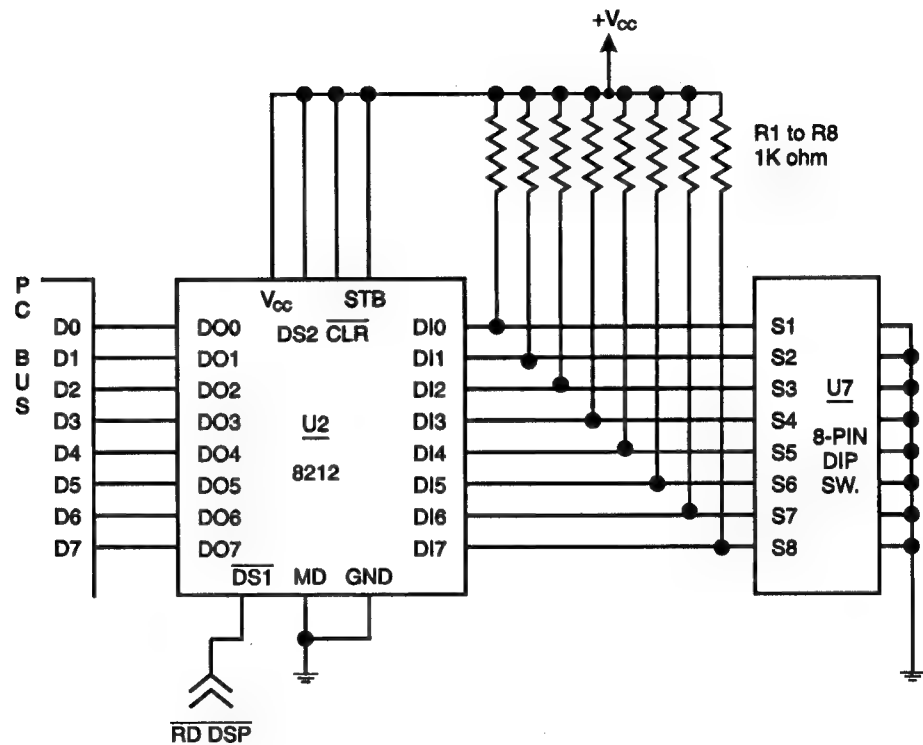
It should be noted that a read from a nonexistent port will put the number FF in the register. An open has the same effect as a high when input to most TTL devices. Since all input devices are tri-states, and none of them are enabled, the data bus is floating (open). When the register latches the information on the data bus, the opens act like highs, so FF is the number latched into the register.

## LAB CIRCUIT

This lab will introduce you to interfacing basic I/O devices to a microprocessor. The decoding circuit shown in Figure 7-19 consists of a dual 2 of 4 decoder (74139), 4 OR gates (7432), and an 8 bit comparator (74688). This circuit will generate 4 read (input) device select pulses (DSP) and 4 write (output) DSPs. These DSPs will be used to clock input and output devices. The input device is an 8212 fed by an 8 position DIP switch, as in Figure 7-20. The output is also an 8212, which feeds a 7 segment decoder and display circuit, as in Figure 7-21. Use the circuit in Figure 7-2 to develop the MEMRD and MEMWR control signals.

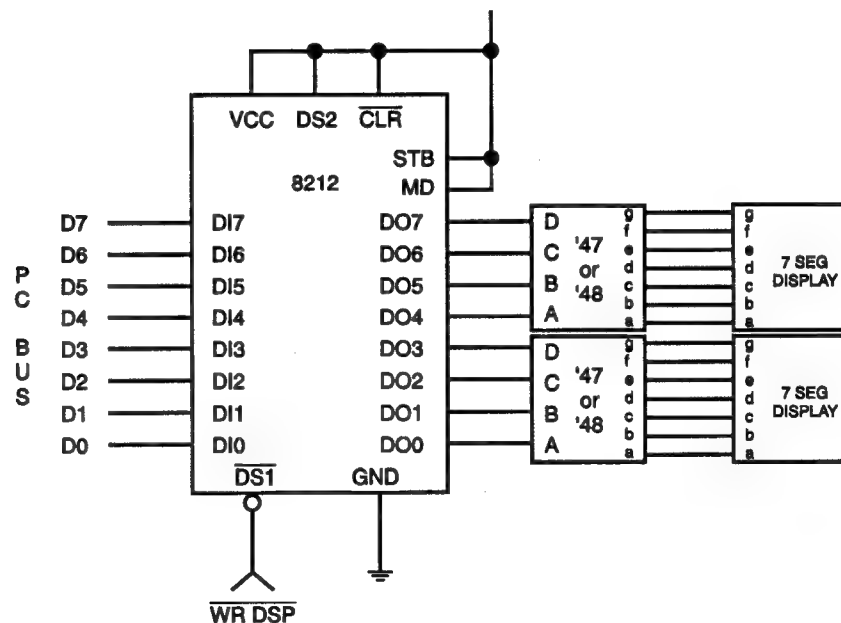
**FIGURE 7-19 I/O Addressing Circuit**



**FIGURE 7-20 I/O Input Circuit**

The 8212 has 8 D-latches, which are the input to 8 tri-states. An input pin, called “mode select” (MD), determines whether the D-latches or the tri-states are clocked by the device select input (DS1, DS2). The ones not clocked are in a constant enable mode, which makes them appear transparent.

The program for the exercise portion of the labs is required to check the DIP switches on the input port every 500 ms. The program must react to any changes between inputs. When the CPU checks the port status (change or no change) in this manner, it is called **polling**. The CPU must input the status, determine if there are any changes, then proceed accordingly.

**FIGURE 7-21 I/O Output Circuit**

---

## CIRCUIT OPERATION

---

This decoding circuit provides 4 input DSPs ( $\overline{RD0300}$ - $\overline{RD0303}$ ) and 4 output DSPs ( $\overline{WR0300}$ - $\overline{WR0303}$ ). The DSP activated depends on the address used when a memory read or write instruction is executed. The input DSPs will be used to clock the tri-states in the 8212 connected as the input device. The output DSPs will be used to clock the D-latches in the output 8212.

---

## LAB PROCEDURE

---

1. Assemble the I/O addressing circuit. *Be sure the computer is off!* Connect the circuit to the extension bus on the trainer.
2. To check the output circuit for proper operation, connect the DSP  $\overline{WR0300}$  to  $\overline{DS1}$  on the output 8212. Next, load and run the following program.

```
LDX      0300      (IMM)
LDA      A,88      (IMM)
STA      A,00      (IND)
```

After the program is executed, the 7 segment display should read "88."

3. To check the input circuit, replace the LDA A,88 (IMM) instruction with LDA A,00 (IND) and add a BRA back to the input instruction after STA A (IND). Connect the DSP  $\overline{RD0300}$  to  $\overline{DS1}$  on the input 8212. When the program is executed, the output port should display the number on the input port.
  4. Change the offset in the indexed instructions to 01 and change the DSP used to clock the 8212s to  $\overline{WR0301}$  and  $\overline{RD0301}$ . Run the program and the output port should again display the number on the input port.
-



## **LAB 7 INPUT/OUTPUT ADDRESSING**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### **QUESTIONS**

---

1. What signals are used to develop a device select pulse?
2. Why is the address bus decoded?
3. Give 3 uses for output DSPs.
4. Why are tri-states used for input interface?
5. Draw a decoding circuit to decode the range of addresses from 0220–0227.

### **EXERCISE**

---

1. Write a program to do the following.
  - a. Have the CPU poll the input port every 500ms. Modify the delay program from Lab 4 for the time delay.
  - b. If the number on the input switches is FF, the output port will display 88. If the number on the input switches is less than FF, the display will be 11.
  - c. Repeat the operation from the beginning.
2. Draw the flowchart and show the registers used.
3. Write the source program and assemble it into machine code.
4. Load your program and check for proper operation by running it.



## LAB

# 8

# I/O COUNTER

## OUTLINE

BCD Instructions

## KEY TERMS

Binary Coded Decimal

Decimal Adjust



Upon completion of this lab, you should know:

1. How to write a program that counts in BCD.
2. How a program uses input data.
3. The interaction between hardware and software.
4. How to write a program that makes decisions based on input data.

## DISCUSSION

### BCD Instructions

Many functions of microprocessor systems require people to read and input numbers. Most people have a difficult time working with hex and binary numbers, so the numbers must be decimal. Usually the digits are encoded and decoded in **Binary Coded Decimal (BCD)** form.

To avoid the need for conversion between BCD and binary, the 6800 has the ability to perform BCD arithmetic on ADD operations.

When an ADD to the accumulator instruction is followed by a **decimal adjust** instruction, the result of the arithmetic is adjusted for a BCD operation. The form and description of the instruction follow.

MNEMONIC	DESCRIPTION
DAA	Decimal Adjust for Addition: the sum in ACCA is adjusted to packed BCD format

It should be noted that the decimal adjust uses the carry and aux carry flags. The decimal adjust instruction *must* follow an ADD instruction. An INC does not affect the carry flag. Also, the destination operand of the ADD instruction *must* be the ACCA.

## LAB PROCEDURE

1. Enter the following instructions into memory.

Mnemonic	Comment
LDA A,34 (IMM)	; Move the packed BCD number 34 to ACCA
ADD A,13 (IMM)	; Add BCD 13 to ACCA
DAA	; Decimal adjust for addition
ADD A,23 (IMM)	; Add BCD 23 to ACCA
DAA	; Decimal adjust for addition
ADD A,88 (IMM)	; Add BCD 88 to ACCA
DAA	; Decimal adjust for addition
ADD A,44 (IMM)	; Add BCD 44 to ACCA
DAA	; Decimal adjust for addition

2. Single step through each instruction, recording the requested registers and flags.

**Instruction 1:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 2:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 3:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 4:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 5:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 6:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 7:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 8:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

**Instruction 9:**

ACCA \_\_\_\_\_ CY \_\_\_\_\_ AC \_\_\_\_\_

## LAB 8 I/O COUNTER

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

1. How does the ALU know when to adjust the lower nibble?
2. How does the ALU know when to adjust the higher nibble?
3. What is the largest BCD number that ACCA can store?
4. What is the advantage of using BCD numbers over binary?
5. What would occur when the following instruction sequence is executed?

LDA B,92 (IMM)

ADD B,08 (IMM)

DAA

### EXERCISE

---

*Using the same hardware from Lab 7, write a flowchart and program that will do the following.*

1. The CPU must cause an internal counter to count from 00d to 59d continuously in 1 second increments.
2. The count must be in decimal.
3. Each number of the count must be displayed on the 7 segment displays of the output port for 1 second.

4. When the count reaches the number set at the input port (on the DIP switches), that number must be displayed on the output port. The count must then return back to 00d and start over.
5. The CPU must sense and react to any change at the input port while the program is running.
  - a. If the input port has a number less than the count, the counter must reset to 00d and restart.
  - b. If the input port has a number greater than the count, the counter should continue as in step 4.
6. Write a program from your flowchart.
7. Assemble the program into machine code.
8. Load and run your program to ensure proper operation.

## LAB

# 9

# INTERRUPTS

## OUTLINE

Vector Address

## KEY TERMS

Clear Interrupt Mask (CLI)

Interrupt Acknowledge (INTA)

Interrupt Mask Bit

Interrupt Request

Maskable Interrupt

Reset Vector Address

Set Interrupt Mask (SEI)

Software Interrupt (SWI)

Vector Address

Vectoring

Vector Table



Upon completion of this lab, you should know:

1. How to use an interrupt vector address.
2. How to generate a hardware interrupt.
3. How the 6800 processes interrupts.

## DISCUSSION

### Vector Address

Interrupts are in effect a jump to a subroutine. But instead of specifying the address with the op code, as with a JSR, the address of the subroutine is stored in 2 memory locations. When the CPU processes an interrupt, it will read the contents of the 2 memory locations and put the numbers into the PC. The next op code fetch will come from the new address in the PC.

This method of getting the address of the interrupt subroutine is called **vectoring**. The memory map of the 6800's vector addresses is shown in Figure 9-1. Note the 4 conditions that use the vector addresses.

For example, if a hardware interrupt occurred and memory contained the numbers FFF8 03 and FFF9 00, the next op code fetch would come from 0300. The program (subroutine) at 0300 would be designed to handle whatever generated the interrupt.

**FIGURE 9-1 6800 Vector Addresses**

FFF8	high byte	} hardware interrupt vector
FFF9	low byte	
FFFA	high byte	} software interrupt vector
FFFB	low byte	
FFFC	high byte	} non-maskable interrupt vector
FFFD	low byte	
FFFE	high byte	} reset vector
FFFF	low byte	

The 6800 also has a software interrupt that uses memory locations FFFA and FFFB to store the vector address of the subroutine. The software interrupt is generated with the instruction **Software Interrupt (SWI)**. The software interrupt can be generated only with the SWI instruction as part of the program.

The part of memory used to store vector address must be stored in non-volatile ROM circuits. This is due to the **Reset Vector Address** stored at FFFE and FFFF.

Whenever the microprocessor is reset, or when it is first powered up, it will read the vector address from FFFE and FFFF. The vector address stored there will be the starting address for the boot-up program.

An interrupt, like a jump to subroutine, will save the return address (PC) on the stack. An interrupt, though, will also save the contents of the other internal registers on the stack. The order of how the registers are stored in memory is shown following.

SP	PCL
SP-1	PCH
SP-2	IXL
SP-3	IXH
SP-4	ACCA
SP-5	ACCB
SP-6	CC
SP-7	Next push (if any)

If the SP contained 03FE, for example, the low byte of the return address (PCL) would be stored at 03FE, the high byte (PCH) at 03FE (SP-1), and the condition codes (CC) at 03F9 (SP-6). The SP would be 03F8 (SP-7) after the pushes.

Because an interrupt saves *all* the internal registers on the stack, not just the return PC, a return from subroutine (RTS) would not work for returning from an interrupt. The RTS instruction pulls only the top 2 bytes of the stack and places the numbers into the PC. The top of the stack after an interrupt contains the old CC and ACCB, not the return PC address. The 6800 has a return from interrupt (RTI) instruction to end an interrupt



subroutine. The RTI instruction pulls all 8 bytes from the stack and returns them to the internal registers. The RTI instruction must *always* end an interrupt subroutine.

Before a **maskable interrupt** will be acknowledged by the CPU, the **interrupt mask bit**, in the condition code register, must be cleared. The interrupt mask bit is cleared with the instruction **Clear Interrupt Mask (CLI)**. Setting the interrupt mask bit with the instruction **Set Interrupt Mask (SEI)** will disable the maskable interrupt input to the CPU.

A hardware interrupt is requested when the interrupt request (IRQ) input is taken low (pin 4 on the 6800 chip). If the interrupt mask bit is cleared, the CPU will enter an interrupt sequence *after* the present instruction is completed.

The first step in the interrupt sequence is to set the interrupt mask bit. This prevents another device from interrupting the CPU while the present interrupt is in progress. After the interrupt mask bit is set, the CPU will save the internal registers on the stack. This is accomplished with 7 consecutive writes to memory, addressed with the SP. The SP is decremented after each write, as illustrated previously.

After the stack writes, the CPU will perform 2 memory reads from FFF8 and FFF9 (see vector addresses Figure 9-1). The numbers at these locations are then loaded into the PC. The next instruction fetch will come from the new address in the PC.

#### Interrupt Sequence

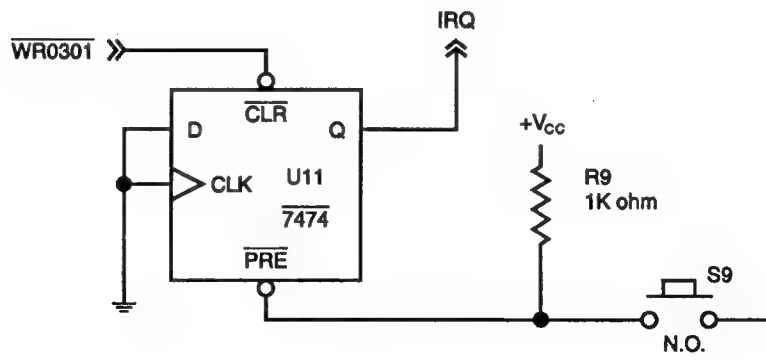
- Step 1. Set interrupt mask bit
- Step 2. Save internal registers on stack
- Step 3. Get subroutine address from FFF8 and FFF9 and put into PC
- Continue normal program sequence

The RTI instruction at the end of the interrupt subroutine would pull the internal registers off the stack. The next op code fetch would be the instruction in the main program after the point where the interrupt occurred.

The NMI and software interrupt would use the same sequence as the maskable interrupt just described. The only difference would be the address at which the vector address is stored. The vector address itself would be determined by the ROM chip in the trainer.

Because the location of the vector addresses of reset and interrupt routines are set in ROM, they cannot be changed. Most trainers, however, use the ROM vectors to run a program that reads an address from R/W memory as the address of the interrupt service routine. The address in the R/W memory location is loaded into the PC. This can be considered as "a vector to a vector to a service routine" instead of "a vector to a subroutine."

**FIGURE 9-2** Interrupt Circuit for Lab 9



## LAB PROCEDURE

1. Add the D-latch and push buttons in Figure 9-2 to the circuit from Lab 8. (Don't forget ground connections.)
2. Connect DSP  $\overline{RD0300}$  to DS1 on the output 8212 and DSP  $\overline{WR0301}$  to the CL input of the D-latch. Be sure the D-latch is cleared ( $Q=\text{low}$ ) initially.
3. Assemble Program 9-1 into machine code.
4. Load and run the program. Your 7 segment display should be displaying a 1 second count.
5. Press the button for IRQ. The display should count to 60d in .1 second intervals, then return to the 1 second count where it left off.

### Program 9-1:

START:

```

0000                ; Beginning address of main program
LDA A,00 (IMM)      ; High byte of int routine address
STA A,E4C3 (EXT)    ; Store in R/W vector location
LDA A,50 (IMM)      ; Low byte of int routine address
STA A,E43D (EXT)    ; Store in R/W vector locations

```

MAIN:

```

LDA A,00 (IMM)      ; Set start count to 00

```

LOOP1:

```

STA A,0300 (EXT)    ; Output count to port 0300
LDA B,0A (IMM)      ; Set length of time delay (1 sec)
BSR Delay (REL)     ; Call time delay
ADD A,01 (IMM)      ; Increment counter
DAA (IMP)           ; Decimal adjust count
BRA Loop1 (REL)     ; Next count

```

ENDMAIN:

INTR:

```

0050                ; Beginning address
                    ; Of the interrupt routine
STA A,0301 (EXT)    ; Clear interrupt request D-latch
LDA A,00            ; Set start count to 00

```

LOOP2:

```

STA A,0300 (EXT)    ; Output count to port 0300
LDA B,01 (IMM)      ; Set length of time delay (.1sec)
BSR Delay (REL)     ; Call time delay
ADD A,01 (IMM)      ; Increment counter
DAA                 ; Decimal adjust counter
CMP A,61 (IMM)      ; Has count reached 61
BNE Loop2 (REL)     ; If not, continue count
RTI (IMP)           ; Return from interrupt

```

ENDINT:

DELAY:

```

PSH A (IMP)         ; Save ACCA contents

```

LOOP3:

```

LDA A,N (IMM)       ; Use value of N for .1sec delay (from Lab 4)

```

## LOOP4:

LDX 20 (DIR)	; Delay instruction
DEC A (IMP)	; Decrement inner loop
BNE Loop4 (REL)	; If not zero, continue inner loop
DEC B (IMP)	; Increment outer loop
BNE Loop3 (REL)	; If not zero, continue delay
PUL A	; Restore ACCA
RTS	; Return to calling program

---



## **LAB 9 INTERRUPTS**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### **QUESTIONS**

---

1. Will the interrupt request bit be set or cleared after returning from an interrupt?
2. Explain the interrupt sequence.
3. What must be cleared before the 6800 will acknowledge an interrupt request?
4. What is the difference between a software and hardware interrupt?
5. How is the interrupt acknowledged in the lab procedure?

### **EXERCISE**

---

1. Write a new interrupt service routine (ISR) to input the value on the input port DIP switches. If the value input is FF, the ISR must count from 0 to 60 in .1sec increments. If the value input is not FF, the ISR must count in .5sec increments.



# LAB 10

## HARDWARE INTERRUPT 2

### OUTLINE

#### Interrupt-driven Systems



Upon completion of this lab, you should know:

1. How an I/O device generates an interrupt.
2. The interaction between I/O devices and interrupt service routines.
3. How to use a microprocessor to monitor the security in a building.

### DISCUSSION

#### Interrupt-driven Systems

Many devices in a microprocessor system need attention or service from the CPU at very infrequent intervals. The CPU must determine somehow when the device needs to be serviced. One method sometimes used requires the CPU to poll the device to see if it needs the CPU's attention, as done in Labs 7 and 8. The drawback to polling devices is the time required for the CPU to determine the device's status.

One way to increase the efficiency of a system is to use interrupts in place of polling. Rather than polling the device to determine its status, the device interrupts the CPU when service is needed. This way the CPU spends time on the device *only* when it needs attention. The rest of the time, the CPU is free to perform other tasks.

A good example of an interrupt-driven system is a burglar alarm for a building. Normally, break-ins would occur at infrequent intervals. Rather than polling each door, window, and so on to determine if the item is open or closed, the hardware would interrupt the CPU if any item were opened. The CPU would then input the status of all the items and determine which was opened.

This lab will simulate a monitoring system using a microprocessor. The lab uses 8 switches, shown in Figure 10-1, to represent the status of a door: 0=open, 1=closed. When a door (switch) is opened, an interrupt request is generated. This will cause the CPU to execute an interrupt routine to service the hardware monitoring the doors.

The service program must input the status of the doors and determine which door is open. Since an individual bit represents the status of a door,

the CPU must determine the state of each bit. A good way to test 1 bit at a time is to use rotate or shift instructions along with the carry flag.

### Program 10-1:

For example, the instruction sequence:

```

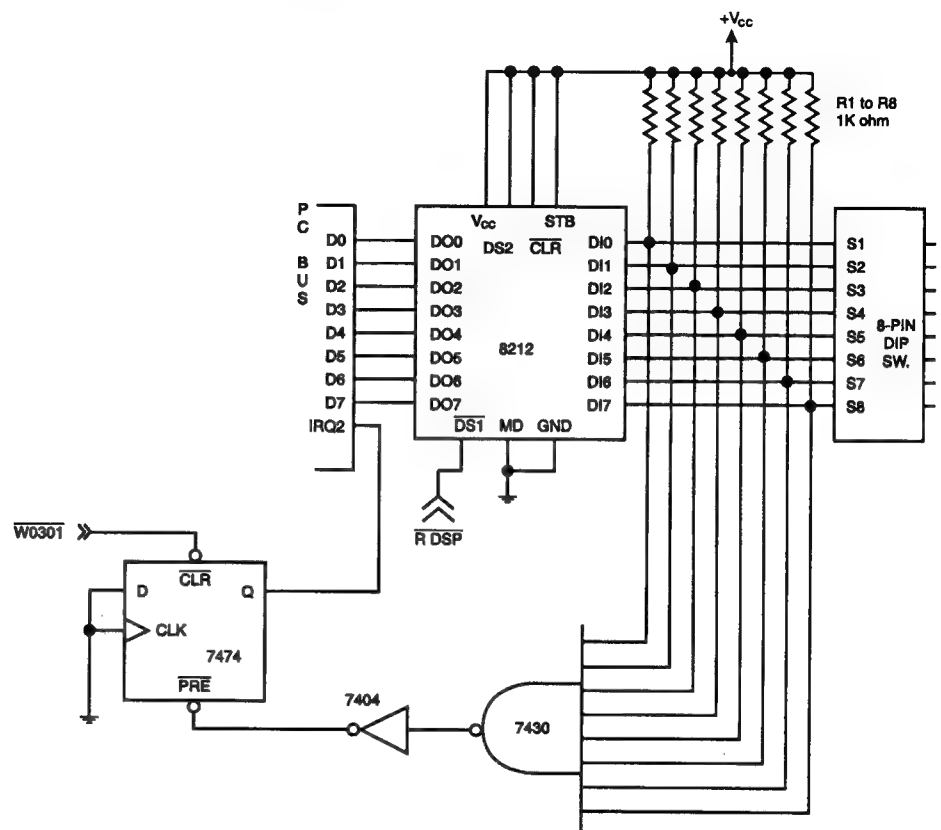
        LDA A,20 (DIR)    ; Get door status byte (port 20)
        LDA B,00 (IMM)    ; Clear count register
Repeat: INC B              ; ACCB contains # of door tested
        ROR A              ; Rotate D0 into carry flag
        BCS Repeat         ; Repeat until a 0 is detected
  
```

will detect a 0 (door open) in the status byte. The number in ACCB will correspond to the open door (D0=door 1). This sequence assumes that there is an open door (a low bit) in the status byte. In addition, it will find only 1 open door. Adding 30 to the number in ACCB would convert the open door number to ASCII code for display or printing.

## LAB PROCEDURE

1. Modify the circuit from Lab 9 as shown in Figure 10-1.

**FIGURE 10-1 Interrupt Circuit for Lab 10**





## LAB 10 HARDWARE INTERRUPT 2

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

1. Which makes more efficient use of the CPU's time, polling or interrupts?
2. Explain how the hardware in this lab generates an interrupt.
3. How many times will the instruction ROR A in Program 10-1 repeat if Door 4 is open?
4. What could be changed in Program 10-1 to cause ACCB to indicate the bit position (D0→D7) of the open door?

### EXERCISE

---

1. Using parts of programs from previous labs, construct a new program to do the following.
  - a. Display a continuous 1 second 0 to 59 BCD count on a 7 segment display.
  - b. When a door (switch) is opened, an interrupt will be generated. The interrupt service routine will determine which door is open; display the open door number for 10 seconds; and resume counting from the point interrupted.



## LAB

# 11

# PROGRAMMABLE PERIPHERAL INTERFACE (8255)

## OUTLINE

Programmable Peripheral Interface

Control Word

## KEY TERMS

Bit Set/Reset Function  
Mode and Port Configuration

Programmable Peripheral  
Interface (PPI)



Upon completion of this lab, you should know:

1. How to program the 8255 for different input/output configurations.
2. The use of the 8255 bit set/reset function.

## DISCUSSION

### Programmable Peripheral Interface

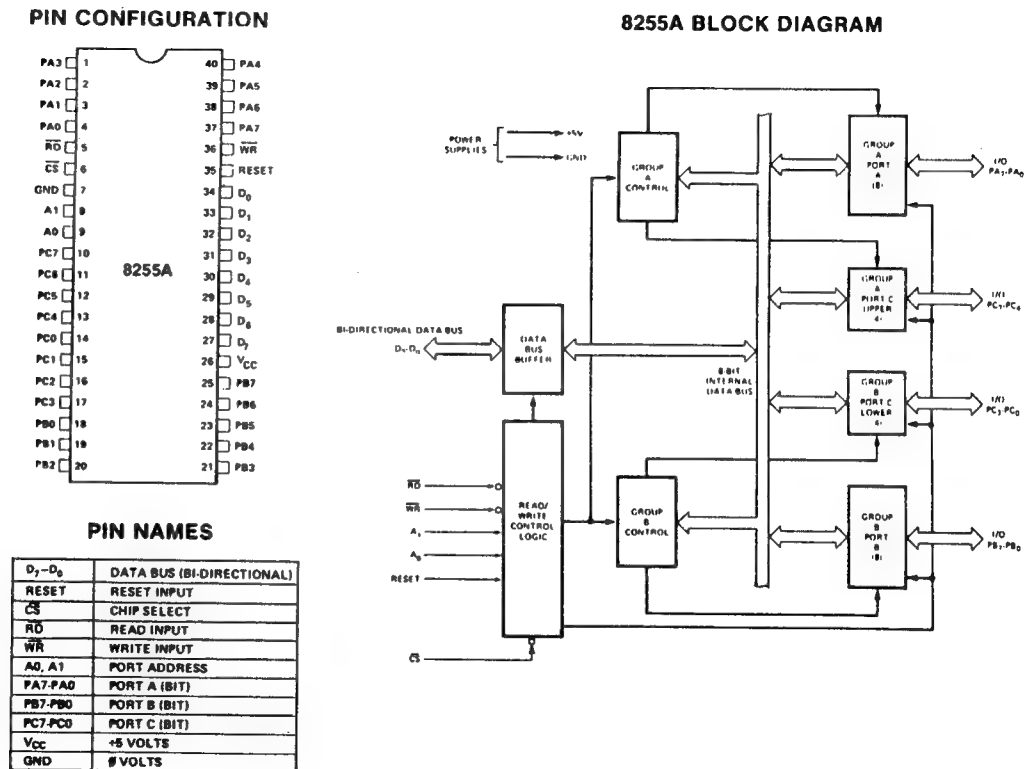
There are many different ICs designed to simplify the interfaces in a microprocessor system. Some must be programmed before they can operate correctly. One of the more common programmable chips is the Intel 8255 **Programmable Peripheral Interface (PPI)**. The 8255 consists of 3 8-bit ports, A, B, and C. The ports can be programmed in a combination of input and output configurations in 3 different modes of operation. In this lab, only 1 mode will be used, mode 0. The 8255 block diagram is shown in Figure 11-1.

The operation of the 8255 is determined by the CPU writing a control word to the control block. The control logic then activates the control lines to configure the ports as input or output. Effectively, each port contains 8 tri-states and 8 D-latches, much like an 8212.

The 8255 contains 4 addressable registers, ports A, B, and C, and the control register. The IC has the logic in the control block to decode 2 address lines into 4 DSPs, similar to the decoding circuit used in the last labs. The 8255 contains a circuit equivalent to the 74139 and 7432 part of the decoding circuit. Those ICs won't be needed for I/O decoding in remaining labs.

The upper part of the address, however, must still be decoded. The 74688 will still be used. The output of the address decoder will be used as

**FIGURE 11-1**  
**8255A/8255A-5**  
**Programmable**  
**Peripheral**  
**Interface**



the chip select (CS) for the 8255. The addresses of the 8255 registers will be in the same range as the port addresses of the last labs as shown.

Address	8255 Register
PORT 0300 =	PORT A
PORT 0301 =	PORT B
PORT 0302 =	PORT C
PORT 0303 =	Control register

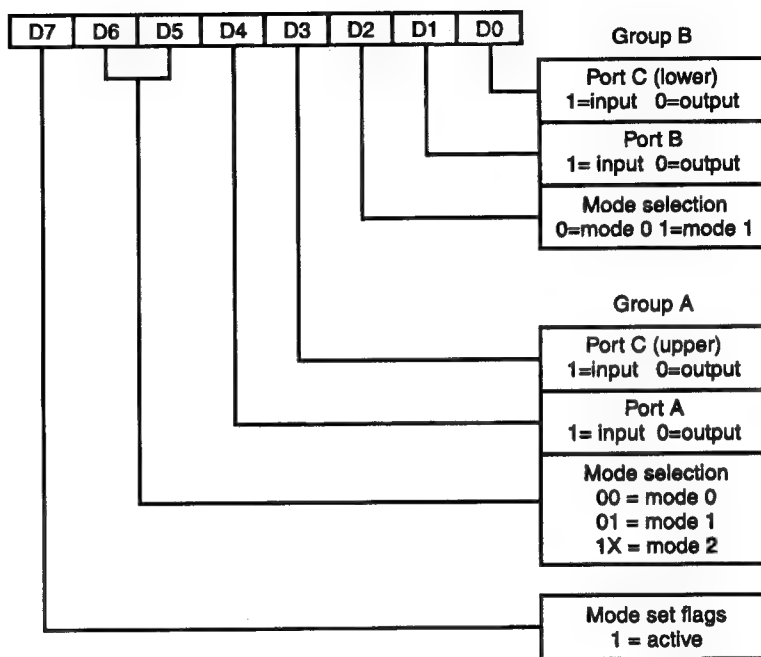
When the CS input is low, the control block decodes A<sub>1</sub> and A<sub>0</sub>, gates the decoded address with the RD or WR control lines, and activates the addressed circuit. It should be noted that the control register is a *write only* register. It is an illegal condition to read from the control register.

## CONTROL WORD

In Mode 0, the C port can be configured as 2 separate 4 bit ports independent of each other. Both halves, however, have the same address. If half is input and the other half output, an input from port C would produce an F (all 1s) in the output nibble. An output to the C port is ignored by the input nibble. The format for the control word is shown in Figure 11-2.

As can be seen, each bit of the control word determines the operation of a different port. In mode 0, each port acts independently of the others. In modes 1 and 2, each half of port C is used by ports A and B for control functions, hence the "group A" and "group B" separation in the control word.

The most significant bit of the control word (D<sub>7</sub>) determines whether the control word defines the **mode and port configuration** or the **bit set/reset function** of the C port (discussed next). When D<sub>7</sub> is set (D<sub>7</sub>=1) the control word is a mode definition.

**FIGURE 11-2 8255 Control Word Format**

The next 2 bits (D6 and D5) define the mode for group A (port A and port C upper). This lab will use mode 0: Bits D4 and D3 configure A and C upper. If the bit is set, the respective port is configured as input, and a cleared bit configures the port as output. The least significant 3 bits configure the Group B ports (port B and C lower) as indicated.

If, for example, Ports A and C upper were needed for input, and ports B and C lower needed for output, the necessary control word would be 98H (for mode 0). See Figure 11-3.

When the C port is configured as output, the individual bits can be set or reset without affecting the other bits. This is called the bit set/reset function. It is active when D7 of the control word is a 0. The format of the bit set/reset control word is shown in Figure 11-4.

For example, to set bit D2 of C lower, the control word would be 05H. See Figure 11-5.

The don't care bits D4, D5, and D6 can be any combination of 1s and 0s. It is customary, however, to clear don't care bits. See Figure 11-6.

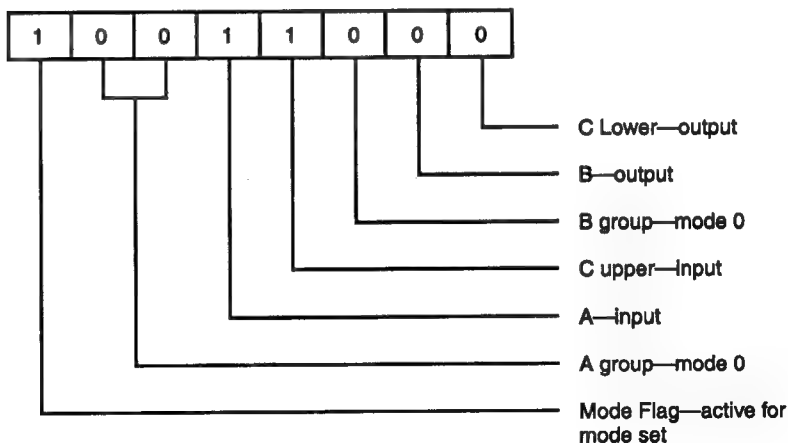
**FIGURE 11-3 Control Word Example**

FIGURE 11-4 Bit Set/Reset Format

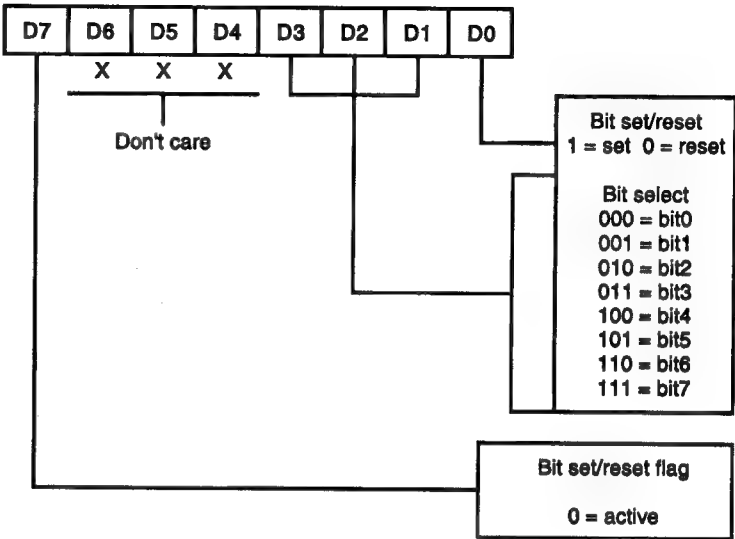


FIGURE 11-5 Bit Set/Reset Example

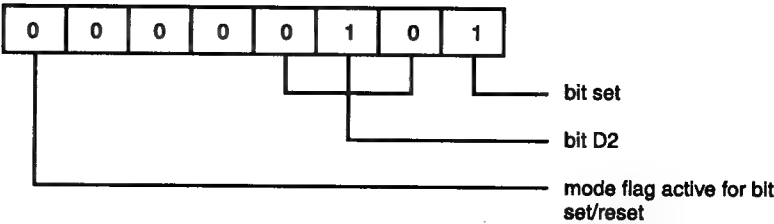
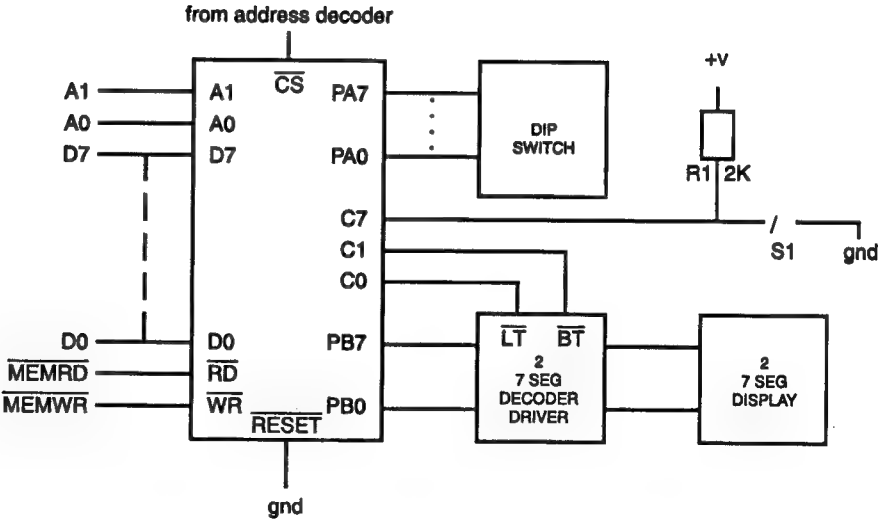


FIGURE 11-6 Circuit for Lab 11



## LAB PROCEDURE

1. Construct the circuit shown below. Use the output of the address decoder for the 8255 chip select.
2. Enter the following program.

**Instruction:**

1	LDA A,98 (IMM)	; Control byte for port configuration
2	STA A,0303 (EXT)	; A & C <sub>U</sub> input, B & C <sub>L</sub> output
3	LDA A,FF (IMM)	; Initial output byte
4	STA A,0302 (EXT)	; Set port C <sub>U</sub> bits
5	STA A,0301 (EXT)	; Set port B bits
Repeat:		
6	LDA A,0300 (EXT)	; Input dip switch setting
7	STA A 0301 (EXT)	; Output switch setting to 7 segment
Cont:		
8	LDA A,0302 (EXT)	; Input switch S1 setting
9	BNE CONT	; Wait for low on switch S1 (C7)
10	LDA A,00 (IMM)	; Control byte to clear bit C0
11	STA A,0303 (EXT)	; Clear bit C0
12	INC A	; Control byte to set bit C0
13	STA A,0303 (EXT)	; Set bit C0
14	BRA Repeat	; Continuous loop

3. Single step through the program with S1 set to a high. Observe the results on the 7 segment display. When instruction 8 is repeated, go to step 4.
4. Change S1 to input a low. Continue single stepping through the program, observing the results on the 7 segment display.
5. Run the program at full speed using different settings of the DIP switches and switch S1.





# **LAB 11 PROGRAMMABLE PERIPHERAL INTERFACE (8255)**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

## **QUESTIONS**

---

1. The byte moved to ACCA in instruction 1, configured the 8255 for A and C<sub>U</sub> input and B and C<sub>L</sub> for output. What control word would reverse the configuration?
2. What condition causes instruction 9 *not* to jump?
3. What control word will clear bit C7?
4. How does the 8255 control section determine whether the control word is a mode set or a bit set/reset function?
5. What did instruction 11 do?

## **EXERCISE**

---

1. Construct a continuous running program that will offer the user a choice of performing a test on the 7 segment display or inputting and displaying a number from the DIP switches. The program will wait for the user to select which part to run from the keyboard. If a 0 is pressed, the 7 segment test will be run, and an 8 selects the display routine.

When running the 7 segment display test, the program must light all segments, then wait for a low on S1. Then the program must blank all segments and wait for a high on S1. After a high is detected on S1, the display must return to its state before running the test routine and returning to the user select part of the program.

The input and display routine must wait for a low on S1 before inputting the state of the DIP switches. The program must then wait for a high on S1 before displaying the results on the 7 segment display and returning to the menu.



# LAB 12

## PARALLEL PRINTER INTERFACE

### OUTLINE

Centronics Printer Interface

### KEY TERMS

BUSY  
Centronics Interface

Handshaking  
INPUT PRIME

STROBE



Upon completion of this lab, you should know:

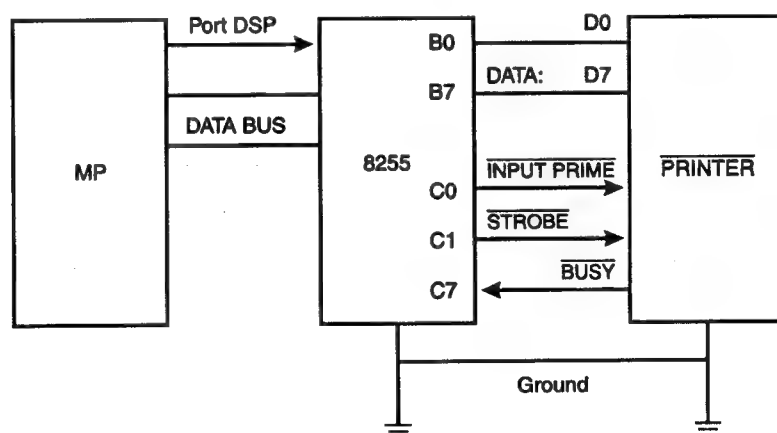
1. The signals used in a centronics printer interface.
2. The definition and use of handshaking.
3. Programming for a centronics interface.

### DISCUSSION

#### Centronics Printer Interface

Printers are often connected to a computer in parallel. A standard printer connection is known as a **centronics interface**, shown in Figure 12-1.

**FIGURE 12-1 Centronics  
Printer Interface**



The connections between the computer and the printer consist of a data path with 8 wires (D0–D7), 3 control lines ( $\overline{\text{INPUT PRIME}}$ ,  $\overline{\text{STROBE}}$ , and  $\overline{\text{BUSY}}$ ), and a common ground connection.

The control lines are used to synchronize the timing of data transfer over the data lines much as S1 was used in Lab 11. The ground connection *must* be common between the pieces of equipment so there is a complete path for current to flow.

The  $\overline{\text{INPUT PRIME}}$  signal performs a reset function on the printer. When the microprocessor outputs a low pulse to the  $\overline{\text{INPUT PRIME}}$ , the printer head moves to the right and any characters in the printer's memory buffer are erased. The printer is now ready to receive characters for printing.

The printer indicates that it is ready for the microprocessor to send characters by placing a high on the  $\overline{\text{BUSY}}$  line. The high on the  $\overline{\text{BUSY}}$  line is how the printer tells the computer, "I'm not busy, send me a character whenever you are ready."

When the computer is ready to print, it will test the  $\overline{\text{BUSY}}$  line to determine if the printer is busy. If a low is detected on the  $\overline{\text{BUSY}}$  line, the computer will wait before sending characters. The computer will keep checking  $\overline{\text{BUSY}}$  for a high before sending any characters to the printer.

Characters are sent to the printer with the  $\overline{\text{STROBE}}$  line and the data lines. The character would be output from the microprocessor to the port corresponding to the printer latch. The latch would apply the character output from the microprocessor to the data lines connected to the printer.

The  $\overline{\text{STROBE}}$  line would be used to clock the data into the buffer of the printer. After the character was output to the data latch, the microprocessor would pulse the  $\overline{\text{STROBE}}$  line to a low. This low would act like a DSP to latch the data into the memory of the printer.

This sequence of checking the  $\overline{\text{BUSY}}$  line, outputting a character, and pulsing the  $\overline{\text{STROBE}}$  line would continue until the printer buffer was full. At that time the printer would start printing. While it was printing, the  $\overline{\text{BUSY}}$  line would be at a low, indicating that the printer was busy.

The control line,  $\overline{\text{BUSY}}$ , allows the microprocessor and the printer to communicate with each other. The printer indicates its status (busy or not busy) by a low or high on the  $\overline{\text{BUSY}}$  line. The microprocessor is asking the status of the printer when it checks the  $\overline{\text{BUSY}}$  line for a high or low.

This type of communication between devices is called **handshaking**. It is like a question-and-answer exchange between devices. The microprocessor checking the high/low status of the  $\overline{\text{BUSY}}$  line asks, "Can I send you data?" The status of the  $\overline{\text{BUSY}}$  line is the answer: high, yes, and low, no.

The low on the  $\overline{\text{STROBE}}$  line is the microprocessor's signal to the printer that there is a valid ASCII character on the data lines. That is the signal to the printer to latch the data into the buffer.

The centronics printer interface requires the printer to have the necessary logic to provide the handshaking. The printer is considered to be an "intelligent" device. Most printers actually have a microprocessor as their controlling device. They are, in effect, a computer with limited functions.

Parallel input from intelligent devices would be similar to the centronics output operation. The timing of the data transfer would be synchronized through the use of handshaking. The data would be input to the computer bus through tri-states, instead of latches as with an output.

## LAB PROCEDURE

1. Remove the 7 segment display, DIP switches, and switch S1 from the Lab 11 circuit.
2. Connect the printer lines to the 8255 as shown in Figure 12-1. Use the port configuration from Lab 11.
3. Enter the following program.

### Instruction

```

1      LDA A,98 (IMM)   ; Control byte for port configuration
2      STA A 0303 (EXT) ; Configure 8255 ports
3      LDA A,03 (IMM)   ; Control byte to set C1 (STROBE)
4      STA A,0303 (EXT) ; Set C1 (STROBE)
5      LDA A,00 (IMM)   ; Control byte to clear C0 (INPUTPRIME)
6      STA A,0303 (EXT) ; Clear C0 (INPUTPRIME)
7      INC A            ; Control byte to set C0
8      STA A,0303 (EXT) ; Set C0
REPEAT:
9      LDA B,2F (DIR)   ; Get print character
BUSY:
10     LDA A,0302 (EXT) ; Input BUSY signal
11     BEQ BUSY         ; Wait for high on BUSY
12     STA B,0301 (EXT) ; Send character to port B
13     LDA A,02 (IMM)   ; Control byte to clear bit C1 (STROBE)
14     STA A, 0303 (EXT) ; Clear bit C1 (STROBE low)
15     INC A            ; Control byte to set bit C1
16     STA A,0303 (EXT) ; Set bit C1
17     BRA REPEAT       ; Send next character

```

4. Use the E command to put 41 in memory location 002F, then single step through the program, stopping at the BRA repeat instruction.
5. Use the E command to change the data at address 002F to 0D. Then continue single stepping, again stopping at BRA repeat.
6. Repeat step 5 using different numbers in location 002F.
7. Remove the paper from the printer and repeat step 6.



## **LAB 12 PARALLEL PRINTER INTERFACE**

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### **QUESTIONS**

---

1. Why are instructions 3 and 4 needed?
2. Which instructions perform handshaking with the printer?
3. Why is handshaking important?
4. What causes the printer to print the printer buffer contents?
5. Which instructions sent the print character to the printer's buffer?
6. Explain the results from step 7.

### **EXERCISE**

---

1. Construct a program to print a series of 7 characters stored in memory.





# LAB 13

## MEMORY INTERFACING

### OUTLINE

Timing

Decoding

### KEY TERMS

Bit Configuration  
Bus Loading  
Decoding

Density  
MEMRD  
MEMWR

Timing  
Unit Loads (UL)



Upon completion of this lab, you should know:

1. Memory address decoding methods.
2. Memory timing requirements.
3. The timing and signals in a memory bus cycle.
4. How memory is tested by software.

### DISCUSSION

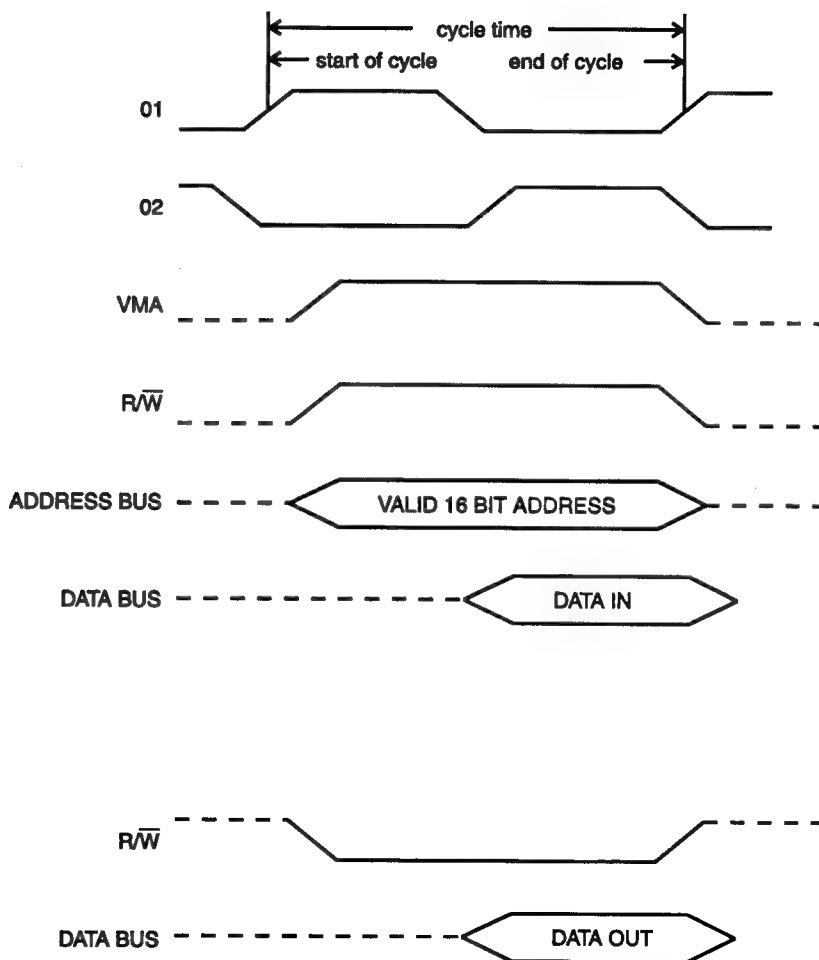
#### Timing

Microprocessor systems employ many different types and combinations of memory ICs. The exact memory arrangement will vary significantly from system to system, depending upon the specific requirements.

The size and type(s) of memory are usually defined by the application of the system. All memory interfaces, though, have three common aspects: **decoding, timing, and bus loading.**

The timing of the data transfers to and from memory is controlled with the MEMRD and MEMWR control lines, as shown in the memory bus cycle timing diagram in Figure 13-1.

ROM ICs need only the MEMRD control line as writes cannot occur. R/W memory ICs must have both read and write control lines connected. A R/W interface is somewhat like an I/O interface. The memory read line times inputs from memory to the CPU, while the memory write control line times outputs from the CPU to memory.

**FIGURE 13-1 Memory Bus Cycle Timing**

## DECODING

Decoding schemes for the memory ICs are also similar to I/O decoding. The main difference is that a large part of the decoding occurs on the memory IC. Just as the 8255 decoded A1 and A0 to address the 4 internal registers, memory ICs decode the address lines necessary to address the number of memory locations on the chip. An IC with 1K addressable locations would have the decoding circuitry to decode 10 address lines ( $2^{10}=1024=1K$ ).

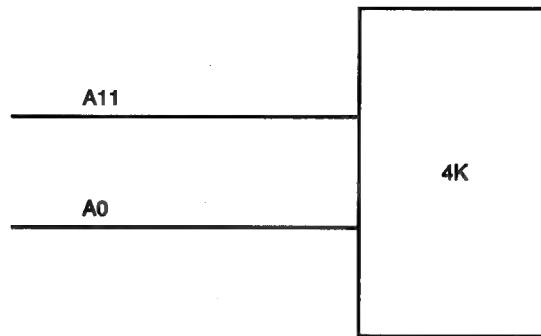
Listed are some address sizes and the required number of address lines.

Address Size	Address Lines
1K	10
2K	11
4K	12
16K	14
64K	16

Notice that as the address size doubles, 1 more address line is needed to address the increased size.

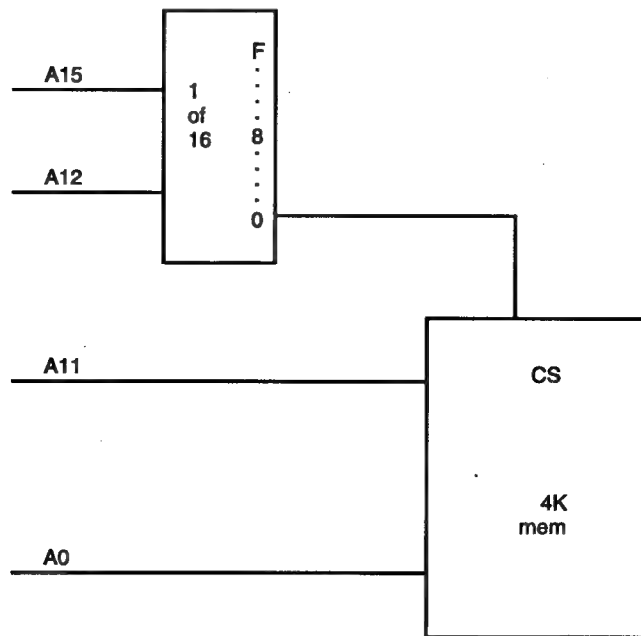
The address lines unused by the memory IC must also be decoded. The arrangement of decoding the unused lines determines the actual address of the memory ICs. For example, using a 4K memory IC would require 12 address lines as shown in Figure 13-2. This would produce addresses of 000 to FFF.

**FIGURE 13-2 12 Address Lines Needed for 4K Memory IC**



The upper 4 address lines (A12–A15) could be decoded by a 1 of 16 decoder. If the 0 output of the decoder were connected to the chip select of the memory IC, as illustrated in Figure 13-3, the range of addresses would be 0000 to 0FFF.

**FIGURE 13-3 Decoding 16 Address Lines**



If the chip select were connected to the 8 output of the decoder, the address range would be 8000 to 8FFF.

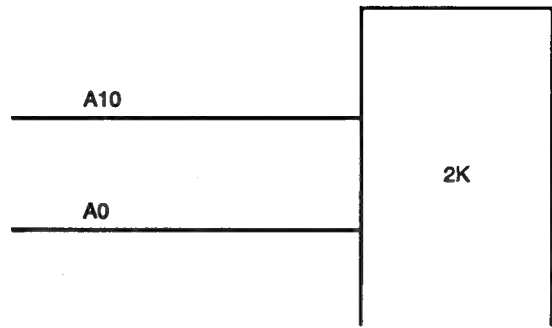
When address decoding falls on 4 bit boundaries, as in the preceding example, it is easy to determine the address in hex. When the decoding is not on 4 bit boundaries, it is usually easier to analyze the addresses in binary and convert to hex if necessary for programming. For example, a 2K memory chip would use 11d address lines as shown in Figure 13-4.

This would produce an address range of:

	000	0000	0000	(000 Hex)	
to:	111	1111	1111	(7FF Hex)	for the memory IC

The decoding for the upper 5 bits would be used as the chip select for the 2K memory. Some possible combinations follow.

**FIGURE 13-4 Addressing 2K Memory IC**

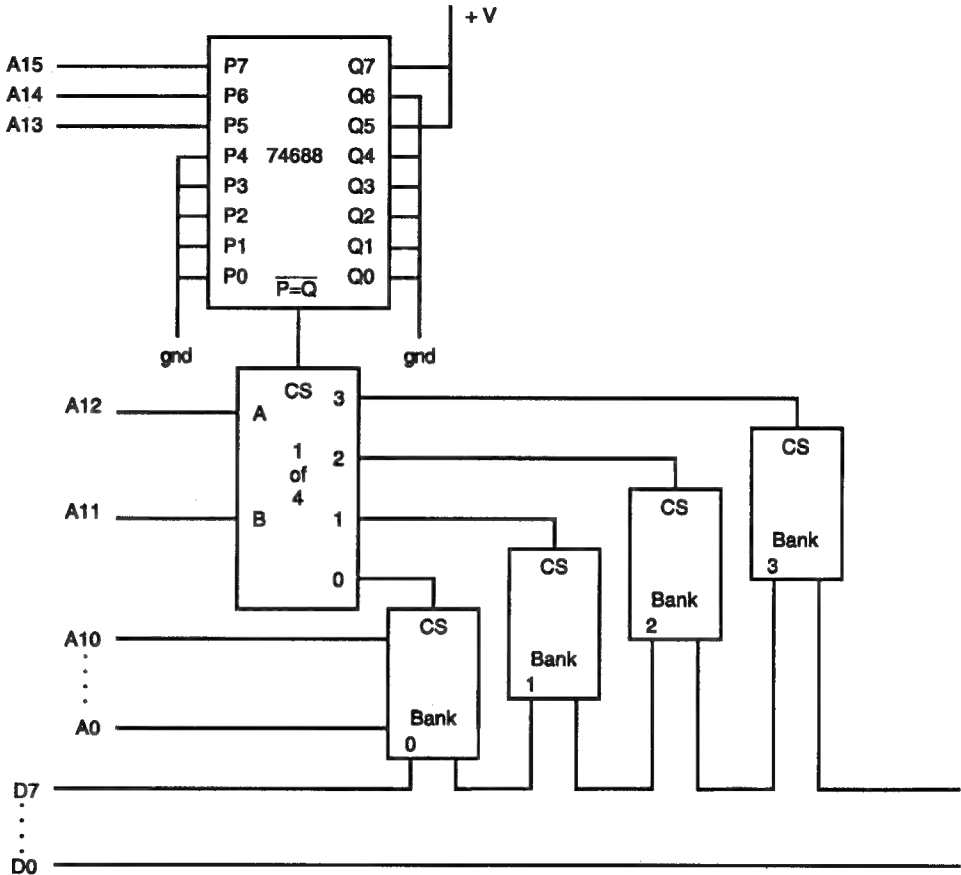


Upper 5 Bits	Lower 11 Bits	Hex Address
1111 1	000 0000 0000	F800
1111 1	111 1111 1111	FFFF
1010 0	000 0000 0000	A000
1010 0	111 1111 1111	A7FF
0000 0	000 0000 0000	0000
0000 0	111 1111 1111	07FF

If more than 2K of memory is needed but only 2K ICs are available, a 1 of X decoder can be used for multiple chip selects. If a 1 of 4 decoder were used to decode A11 and A12, then 4 2K memory ICs can be used to produce 8K of memory.

The following circuit (Figure 13-5) would produce 8K of memory starting at address A000.

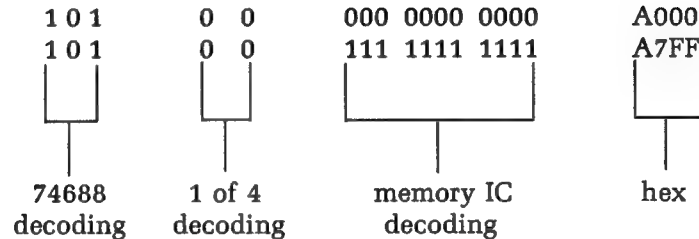
**FIGURE 13-5 8K Memory Interface Using 2K Memory IC's**



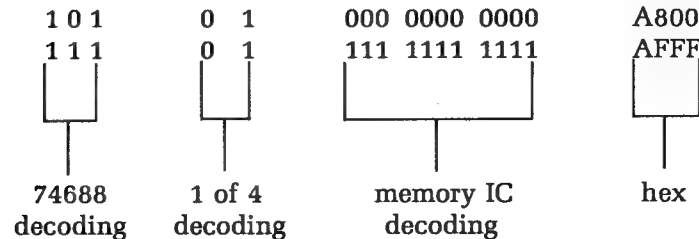
Examining the decoding addresses in binary produces the results shown in Figure 13-6 for bank 0 and bank 1.

**FIGURE 13-6**  
Determining Hex  
Addresses for 8K Memory  
Interface

Bank 0 address range:



Bank 1 address range:



Refer to this circuit when executing this Lab.

## BUS LOADING

The last aspect of memory interfacing is the loading effect the ICs have on the address and data bus. This is primarily determined by the **bit configuration** and **density** of the memory ICs. The bit density of a memory IC is the number of individual memory cells the chip can store. The bit configuration is how the memory cells are accessed. A 16K memory IC, for example, has a density of 16384 memory cells. The configuration could be as follows.

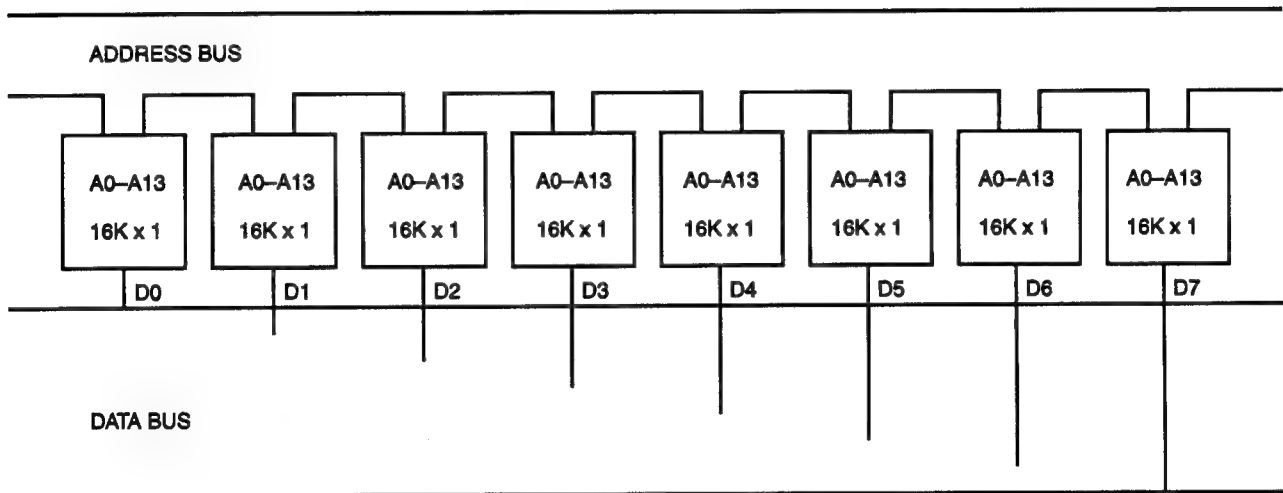
16K x 1  
8K x 2  
4K x 4  
2K x 8

In the 16K x 1 configuration, there would be 16K different addresses with 1 bit stored at each address. The 2K x 8 configuration would have 2K of addresses, but each address would access 8 bits.

Using the 16K x 1 and the 2K x 8 configurations to make a 16K x 8 memory interface will show the difference in loading effects on the busses. The total number of memory cells needed is:  $16K \times 8 = 128K$ .

Since the memory ICs contain 16K memory cells in each configuration, 8 ICs are needed for the total memory. For simplicity, the upper address lines are not included.

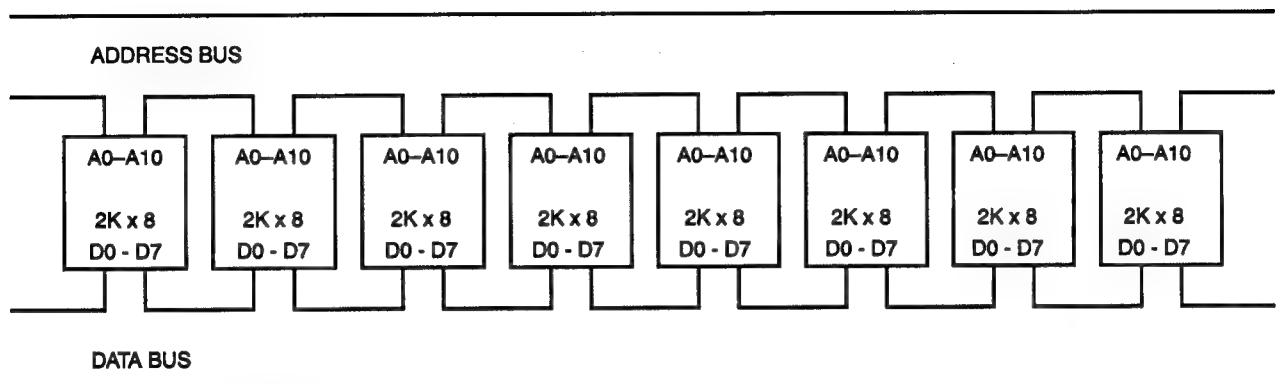
Figure 13-7 shows the use of 8 16K x 1s.



**FIGURE 13-7 Bus Loading for 16K x 1**

Each memory IC must be connected to address lines A0-A13. This produces a total of 8 **Unit Loads (UL)** on these address lines. The data lines, on the other hand, are each connected to only 1 data line. The UL on the data lines is therefore 1.

In the configuration shown in Figure 13-8, address lines A0-A10 have the same UL as in the 16K x 1 configuration: 8. All 8 lines of the data bus are connected to each IC. This produces a UL of 8 on each of the data lines.

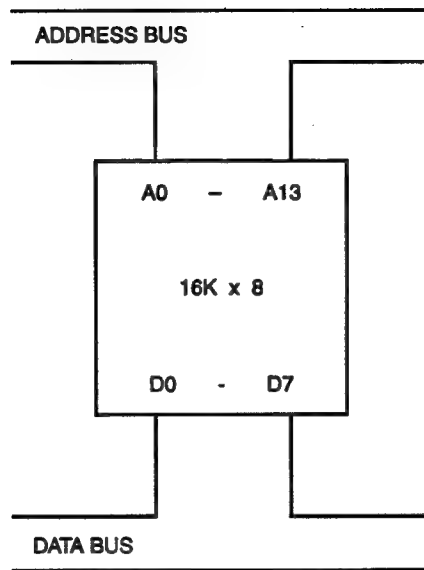


**FIGURE 13-8 Bus Loading for 2K x 8**

The smallest UL on the busses is produced when the entire memory is in 1 IC. If a memory IC with a density of 128K arranged as a 16K x 8 were used instead of 8 16K ICs in the 2 previous examples, the UL on all the address and data lines would be 1.

Figure 13-9 shows the preferred configuration to be used whenever possible.

**FIGURE 13-9 Bus Loading  
for 16K x 8**




---

## LAB PROCEDURE

1. Construct the decoding circuit in Figure 13-5. Use 1 memory IC and leave the chip select unconnected.
  2. Connect the bank 0 output of the decoder to the chip select of the memory IC. This will put the memory address range from A000 to A7FF.
  3. Use the ENTER and DISPLAY commands to verify the memory IC operation.
  4. Change the chip select connection from the decoder's bank 0 output to the bank 3 output.
  5. Repeat step 3 for the new address range.
-





## LAB 13 MEMORY INTERFACING

---

Name: \_\_\_\_\_ Date: \_\_\_\_\_

### QUESTIONS

---

1. What are the address ranges for banks 1 and 2?
2. How many address lines are needed to decode a 32K x 2 memory IC?
3. What is the unit load on the address bus in Figure 13-5?
4. What is the unit load on the data bus in Figure 13-5?
5. What program changes were required for step 5?

### EXERCISE

---

1. Develop a program to load 00 into each memory location of the IC used in this lab. Each location must be read and verified to contain 00. If an error exists, the program must display the address.



# APPENDIX

# A

# 6800 INSTRUCTION SET

The instruction set for the Motorola 6800 consists of 197 individual instructions. The instruction set is divided into 4 groups according to the type of operation, as follows.

1. Jump and branch instructions shown in Table A-1.
2. Index and stack pointer instructions shown in Table A-2.
3. Condition code register (flag) instructions shown in Table A-3.
4. Accumulator and memory operations shown in Table A-4.

														COND. CODE REG.							
OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			IMPLIED			BRANCH TEST	5	4	3	2	1	0	
		OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C	
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•	
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•	
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•	
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•	
Branch If > Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•	
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•	
Branch If Higher	BHI	22	4	2										$C + Z = 0$	•	•	•	•	•	•	
Branch If < Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•	
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	•	•	•	•	•	•	
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•	
Branch If Minus	BMI	28	4	2										N = 1	•	•	•	•	•	•	
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•	
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•	
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•	
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•	
Branch To Subroutine	BSR	8D	8	2										} See Special Operations	•	•	•	•	•	•	
Jump	JMP				6E	4	2	7E	3	3					•	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3				•	•	•	•	•	•	•	
No Operation	NOP										01	2	1	} Advances Prog. Cntr. Only	•	•	•	•	•	•	
Return From Interrupt	RTI										3B	10	1		•	•	•	•	•	•	•
Return From Subroutine	RTS										39	5	1	} See Special Operations	•	•	•	•	•	•	
Software Interrupt	SWI										3F	12	1		•	•	•	•	•	•	•
Wait for Interrupt*	WAI										3E	9	1		•	•	•	•	•	•	•

\*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

- ① (All) Load Condition Code Register from Stack. (See Special Operations)
- ② (Bit 1) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

TABLE A-1 Jump and Branch Instructions

POINTER OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
		OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$X_H - M, X_L - (M + 1)$	•	•	①	1	②	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	1	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	1	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	③	1	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	③	1	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	③	1	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	③	1	R	•
Indx Reg $\rightarrow$ Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr $\rightarrow$ Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

② (Bit V) Test: 2's complement overflow from subtraction of ms bytes?

③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

TABLE A-2 Index Register and Stack Pointer Instructions

b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>
H	I	N	Z	V	C

**H** = Half-carry; set whenever a carry from b<sub>3</sub> to b<sub>4</sub> of the result is generated by ADD, ABA, ADC; cleared if no b<sub>3</sub> to b<sub>4</sub> carry; not affected by other instructions.

**I** = Interrupt Mask; set by hardware or software interrupt or SEI instruction; cleared by CLI instruction. (Normally not used in arithmetic operations.) Restored to a zero as a result of an RTI instruction if I<sub>m</sub> stored on the stack is low.

**N** = Negative; set if high order bit (b<sub>7</sub>) of result is set; cleared otherwise.

**Z** = Zero; set if result = 0; cleared otherwise.

**V** = Overflow; set if there was arithmetic overflow as a result of the operation; cleared otherwise.

**C** = Carry; set if there was a carry from the most significant bit (b<sub>7</sub>) of the result; cleared otherwise.

#### — CONDITION CODE REGISTER INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	COND. CODE REG.					
		OP	~	=		H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	0 $\rightarrow$ C	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	0 $\rightarrow$ I	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 $\rightarrow$ V	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	1 $\rightarrow$ C	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	1 $\rightarrow$ I	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	1 $\rightarrow$ V	•	•	•	•	S	•
Accmtr A $\rightarrow$ CCR	TAP	06	2	1	A $\rightarrow$ CCR	①					
CCR $\rightarrow$ Accmtr A	TPA	07	2	1	CCR $\rightarrow$ A						

R = Reset

S = Set

• = Not affected

① (ALL) Set according to the contents of Accumulator A.

TABLE A-3 Condition Code Register Instructions

## ACCUMULATOR AND MEMORY OPERATIONS

		ADDRESSING MODES					BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
OPERATIONS	MNEMONIC	IMMED	DIRECT	INDEX	EXTND	IMPLIED	(All register labels refer to contents)	5	4	3	2	1	0
		OP ~ =	OP ~ =	OP ~ =	OP ~ =	OP ~ =		H	I	N	Z	V	C
Add	ADDA	9B 2 2	9B 3 2	A8 5 2	BB 4 3		A + M + A	1	1	1	1	1	1
	ADDB	CB 2 2	DB 3 2	E8 5 2	FB 4 3		B + M + B	1	1	1	1	1	1
Add Acmltrs	ABA					1B 2 1	A + B + A	1	1	1	1	1	1
Add with Carry	ADCA	89 2 2	99 3 2	A9 5 2	89 4 3		A + M + C + A	1	1	1	1	1	1
	ADCB	C9 2 2	D9 3 2	E9 5 2	F9 4 3		B + M + C + B	1	1	1	1	1	1
And	ANDA	84 2 2	94 3 2	A4 5 2	84 4 3		A - M + A	1	1	1	1	1	1
	ANDB	C4 2 2	D4 3 2	E4 5 2	F4 4 3		B - M + B	1	1	1	1	1	1
Bit Test	BITA	85 2 2	95 3 2	A5 5 2	85 4 3		A - M	1	1	1	1	1	1
	BITB	C5 2 2	D5 3 2	E5 5 2	F5 4 3		B - M	1	1	1	1	1	1
Clear	CLR			6F 7 2	7F 6 3		00 - M	1	1	1	1	1	1
	CLRA					4F 2 1	00 - A	1	1	1	1	1	1
	CLRB					5F 2 1	00 - B	1	1	1	1	1	1
Compare	CMPA	81 2 2	91 3 2	A1 5 2	B1 4 3		A - M	1	1	1	1	1	1
	CMPB	C1 2 2	D1 3 2	E1 5 2	F1 4 3		B - M	1	1	1	1	1	1
Compare Acmltrs	CBA					11 2 1	A - B	1	1	1	1	1	1
Complement, 1's	COM			63 7 2	73 6 3		M - M	1	1	1	1	1	1
	COMA					43 2 1	A - A	1	1	1	1	1	1
	COMB					53 2 1	B - B	1	1	1	1	1	1
Complement, 2's (Negate)	NEG			60 7 2	70 6 3		00 - M - M	1	1	1	1	1	1
	NEGA					40 2 1	00 - A - A	1	1	1	1	1	1
	NEGB					50 2 1	00 - B - B	1	1	1	1	1	1
Decimal Adjust, A	DAA					19 2 1	Converts Binary Add. of BCD Characters into BCD Format	1	1	1	1	1	1
Decrement	DEC			6A 7 2	7A 6 3		M - 1 + M	1	1	1	1	1	1
	DECA					4A 2 1	A - 1 + A	1	1	1	1	1	1
	DECB					5A 2 1	B - 1 + B	1	1	1	1	1	1
Exclusive OR	EDRA	88 2 2	98 3 2	A8 5 2	88 4 3		A ⊕ M + A	1	1	1	1	1	1
	EORB	C8 2 2	D8 3 2	E8 5 2	F8 4 3		B ⊕ M + B	1	1	1	1	1	1
Increment	INC			6C 7 2	7C 6 3		M + 1 - M	1	1	1	1	1	1
	INCA					4C 2 1	A + 1 - A	1	1	1	1	1	1
	INCB					5C 2 1	B + 1 - B	1	1	1	1	1	1
Load Acmltr	LDAA	86 2 2	96 3 2	A6 5 2	86 4 3		M - A	1	1	1	1	1	1
	LDAB	C6 2 2	D6 3 2	E6 5 2	F6 4 3		M - B	1	1	1	1	1	1
Or, Inclusive	ORAA	8A 2 2	9A 3 2	AA 5 2	8A 4 3		A + M - A	1	1	1	1	1	1
	ORAB	CA 2 2	DA 3 2	EA 5 2	FA 4 3		B + M - B	1	1	1	1	1	1
Push Data	PSHA					36 4 1	A - Msp, SP - 1 - SP	1	1	1	1	1	1
	PSHB					37 4 1	B - Msp, SP - 1 - SP	1	1	1	1	1	1
Pull Data	PULA					32 4 1	SP + 1 - SP, Msp + A	1	1	1	1	1	1
	PULB					33 4 1	SP + 1 - SP, Msp + B	1	1	1	1	1	1
Rotate Left	ROL			69 7 2	79 6 3		M	1	1	1	1	1	1
	ROLA					49 2 1	A	1	1	1	1	1	1
	ROLB					59 2 1	B	1	1	1	1	1	1
Rotate Right	ROR			66 7 2	76 6 3		M	1	1	1	1	1	1
	RORA					46 2 1	A	1	1	1	1	1	1
	RORB					56 2 1	B	1	1	1	1	1	1
Shift Left, Arithmetic	ASL			68 7 2	78 6 3		M	1	1	1	1	1	1
	ASLA					48 2 1	A	1	1	1	1	1	1
	ASLB					58 2 1	B	1	1	1	1	1	1
Shift Right, Arithmetic	ASR			67 7 2	77 6 3		M	1	1	1	1	1	1
	ASRA					47 2 1	A	1	1	1	1	1	1
	ASRB					57 2 1	B	1	1	1	1	1	1
Shift Right, Logic	LSR			64 7 2	74 6 3		M	1	1	1	1	1	1
	LSRA					44 2 1	A	1	1	1	1	1	1
	LSRB					54 2 1	B	1	1	1	1	1	1
Store Acmltr.	STAA		97 4 2	A7 6 2	B7 5 3		A - M	1	1	1	1	1	1
	STAB		D7 4 2	E7 6 2	F7 5 3		B - M	1	1	1	1	1	1
Subtract	SUBA	80 2 2	90 3 2	A0 5 2	80 4 3		A - M + A	1	1	1	1	1	1
	SUBB	C0 2 2	D0 3 2	E0 5 2	F0 4 3		B - M + B	1	1	1	1	1	1
Subtract Acmltrs.	SBA					10 2 1	A - B + A	1	1	1	1	1	1
Subtr. with Carry	SBCA	82 2 2	92 3 2	A2 5 2	82 4 3		A - M - C + A	1	1	1	1	1	1
	SBCB	C2 2 2	D2 3 2	E2 5 2	F2 4 3		B - M - C + B	1	1	1	1	1	1
Transfer Acmltrs	TAB					16 2 1	A - B	1	1	1	1	1	1
	TBA					17 2 1	B - A	1	1	1	1	1	1
Test, Zero or Minus	TST			6D 7 2	7D 6 3		M - 00	1	1	1	1	1	1
	TSTA					4D 2 1	A - 00	1	1	1	1	1	1
	TSTB					5D 2 1	B - 00	1	1	1	1	1	1

## LEGEND:

OP Operation Code (Hexadecimal);  
 ~ Number of MPU Cycles;  
 # Number of Program Bytes;  
 + Arithmetic Plus;  
 - Arithmetic Minus;  
 • Boolean AND;  
 Msp Contents of memory location pointed to by Stack Pointer;  
 + Boolean Inclusive OR;  
 ⊕ Boolean Exclusive OR;  
 M Complement of M;  
 → Transfer Into;  
 0 Bit = Zero;  
 00 Byte = Zero;

## CONDITION CODE SYMBOLS:

H Half-carry from bit 3;  
 I Interrupt mask;  
 M Negative (sign bit);  
 Z Zero (byte);  
 V Overflow, 2's complement;  
 C Carry from bit 7;  
 R Reset Always;  
 S Set Always;  
 † Test and set if true, cleared otherwise;  
 • Not Affected

## CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

1 (Bit V) Test: Result = 10000000?  
 2 (Bit C) Test: Result = 00000000?  
 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine?  
 (Not cleared if previously set.)  
 4 (Bit V) Test: Operand = 10000000 prior to execution?  
 5 (Bit V) Test: Operand = 01111111 prior to execution?  
 6 (Bit V) Test: Set equal to result of N⊕C after shift has occurred.

Note - Accumulator addressing mode instructions are included in the column for IMPLIED addressing

TABLE A-4 Accumulator and Memory Operations



## ADAPTING TO OTHER TRAINERS

This appendix provides information on adapting the labs to different trainers. If the trainer in use does *not* contain R/W memory (RAM) at address 0000 through 007F, the address of the programs must be changed to an address at which R/W memory is present. The memory locations used in the programs must also be changed to an address where R/W memory is present.

The Elenco XK300 will be used as an example. The prompt is a dash in the left-most 7 segment display. All of the following functions are entered from the prompt.

### **LAB 1** To enter the Display function:

1. Enter the address.  
The 4 left 7 segment displays will display the address.
2. Press M.  
The 2 right 7 segment displays will display the number in the addressed memory location.
3. Press GO to increment the address.
4. Press M to decrement the address.
5. Enter a number to change the value in the addressed memory location.

### To enter the register function:

1. Press RD.  
The right 2 displays indicate the register whose contents are being displayed.  
The contents of the register are displayed on the left 2 or 4 displays.
2. Press GO to see next register.
3. Press M to see last register.
4. Enter a number to change the value in the displayed register.

### To single step through a program:

1. Enter the register function.
2. Set the PC to the starting address of the program.
3. Press T/B to execute the instruction.  
The registers can be viewed and/or changed as described in the register function.

### To run a program at full speed:

1. Enter the starting address of the program.
2. Press GO.

To set a breakpoint:

1. Press FS.
2. Press T/B.
3. Enter address of breakpoint.
4. Press FS.
5. Press EX to return to the prompt.

To clear a breakpoint:

1. Enter breakpoint function (press FS then T/B).  
If a breakpoint is set, the address will be displayed on the 4 left 7 segments.
2. Press FC.

A new breakpoint can now be set (as described above), or press RS or EX to return to the prompt.

**LABS 7-13** Labs 7-13 use various signals generated by the CPU. The maximum load on any line is 2 TTL devices. Most trainers can handle this load without buffering. Some trainers, however, may require buffering.

The port addresses used in Labs 7-12 are 0300-0303 with mirroring at 0310-0313 through 0370-0373. If these addresses conflict with other I/O devices or memory addresses, change the address decoding on the 74688 to unused addresses. The port addresses used in the program must be changed accordingly.

**LABS 9 and 10** These labs use interrupt vectors to address interrupt service routines. The address used in the lab is E43C. If the trainer uses another address as the interrupt vector, the program must be changed to the correct address.

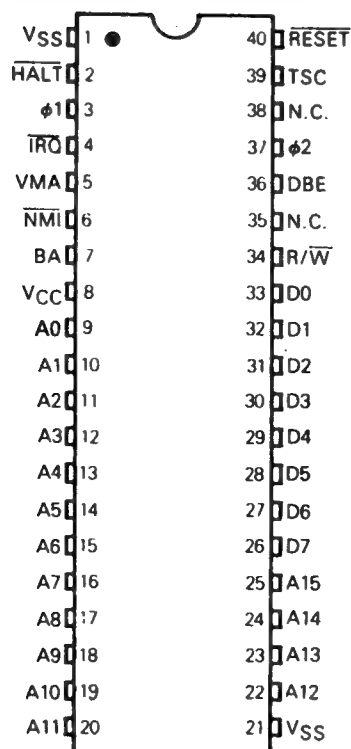
**LAB 13** Lab 13 uses addresses A000-BFFF. If these addresses conflict with I/O ports or memory addresses, change the address decoding on the 74688 to unused addresses.



# APPENDIX

# C

# DATA SHEETS



## ORDERING INFORMATION

Package Type	Frequency (MHz)	Temperature	Order Number
Cerdip S Suffix	1.0	0°C to 70°C	MC6800S
	1.0	-40°C to 85°C	MC6800CS
	1.5	0°C to 70°C	MC68A00S
	1.5	-40°C to 85°C	MC68A00CS
	2.0	0°C to 70°C	MC68B00S
Plastic P Suffix	1.0	0°C to 70°C	MC6800P
	1.0	-40°C to 85°C	MC6800CP
	1.5	0°C to 70°C	MC68A00P
	1.5	-40°C to 85°C	MC68A00CP
	2.0	0°C to 70°C	MC68B00P

FIGURE C-1 MC 6800 Pin Assignment

# SN5404, SN54LS04, SN54S04, SN7404, SN74LS04, SN74S04 HEX INVERTERS

DECEMBER 1983—REVISED MARCH 1988

- Package Options Include Plastic "Small Outline" Packages, Ceramic Chip Carriers and Flat Packages, and Plastic and Ceramic DIPs
- Dependable Texas Instruments Quality and Reliability

## description

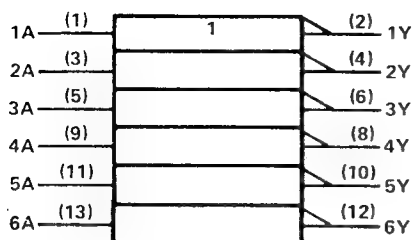
These devices contain six independent inverters.

The SN5404, SN54LS04, and SN54S04 are characterized for operation over the full military temperature range of  $-55^{\circ}\text{C}$  to  $125^{\circ}\text{C}$ . The SN7404, SN74LS04, and SN74S04 are characterized for operation from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .

FUNCTION TABLE (each inverter)

INPUTS	OUTPUT
A	Y
H	L
L	H

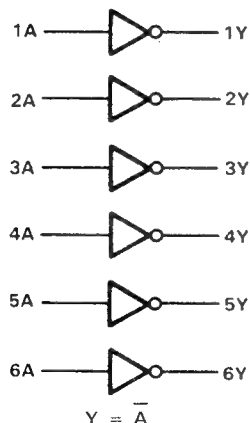
## logic symbol†



†This symbol is in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 617-12.

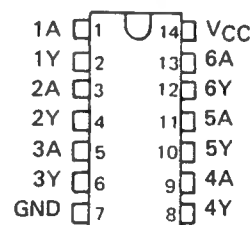
Pin numbers shown are for D, J, and N packages.

## logic diagram (positive logic)



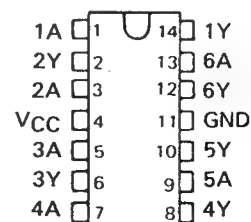
SN5404 . . . J PACKAGE  
SN54LS04, SN54S04 . . . J OR W PACKAGE  
SN7404 . . . N PACKAGE  
SN74LS04, SN74S04 . . . D OR N PACKAGE

(TOP VIEW)



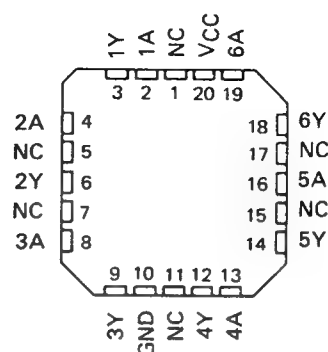
SN5404 . . . W PACKAGE

(TOP VIEW)



SN54LS04, SN54S04 . . . FK PACKAGE

(TOP VIEW)



NC - No internal connection

FIGURE C-2

# **SN5410, SN54LS10, SN54S10, SN7410, SN74LS10, SN74S10** **TRIPLE 3-INPUT POSITIVE-NAND GATES**

DECEMBER 1983—REVISED MARCH 1988

- Package Options Include Plastic "Small Outline" Packages, Ceramic Chip Carriers and Flat Packages, and Plastic and Ceramic DIPs
- Dependable Texas Instruments Quality and Reliability

## description

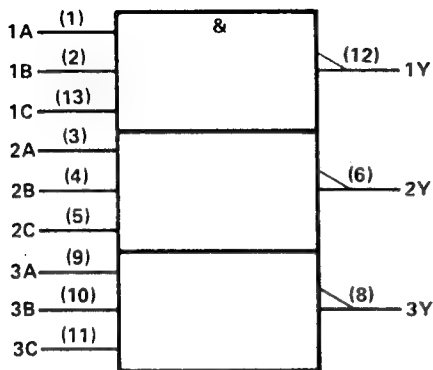
These devices contain three independent 3-input NAND gates.

The SN5410, SN54LS10, and SN54S10 are characterized for operation over the full military temperature range of  $-55^{\circ}\text{C}$  to  $125^{\circ}\text{C}$ . The SN7410, SN74LS10, and SN74S10 are characterized for operation from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .

**FUNCTION TABLE (each gate)**

INPUTS			OUTPUT
A	B	C	Y
H	H	H	L
L	X	X	H
X	L	X	H
X	X	L	H

## logic symbol†



†This symbol is in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 617-12.

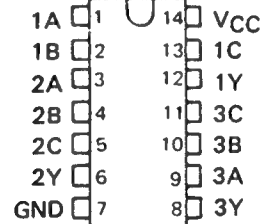
Pin numbers shown are for D, J, and N packages.

## positive logic

$$Y = \overline{A \cdot B \cdot C} \text{ or } Y = \overline{A} + \overline{B} + \overline{C}$$

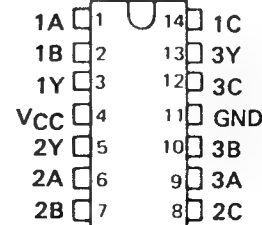
SN5410 . . . J PACKAGE  
SN54LS10, SN54S10 . . . J OR W PACKAGE  
SN7410 . . . N PACKAGE  
SN74LS10, SN74S10 . . . D OR N PACKAGE

**(TOP VIEW)**



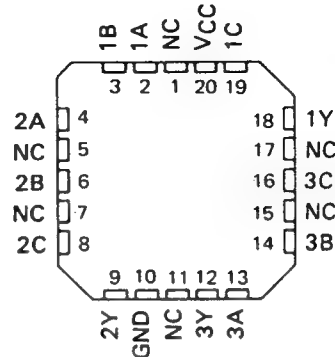
SN5410 . . . W PACKAGE

**(TOP VIEW)**



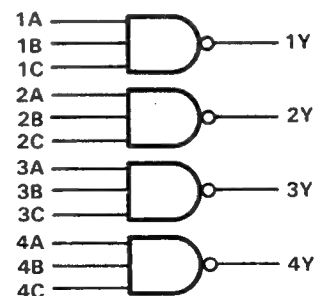
SN54LS10, SN54S10 . . . FK PACKAGE

**(TOP VIEW)**



NC - No internal connection

## logic diagram (positive logic)



**FIGURE C-3**

SN5430, SN54LS30, SN54S30,  
SN7430, SN74LS30, SN74S30  
8-INPUT POSITIVE-NAND GATES

DECEMBER 1983—REVISED MARCH 1988

- Package Options Include Plastic “Small Outline” Packages, Ceramic Chip Carriers and Flat Packages, and Plastic and Ceramic DIPs
- Dependable Texas Instruments Quality and Reliability

description

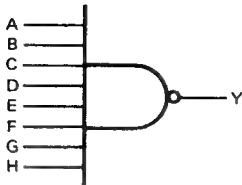
These devices contain a single 8-input NAND gate.

The SN5430, SN54LS30, and SN54S30 are characterized for operation over the full military range of –55 °C to 125 °C. The SN7430, SN74LS30, and SN74S30 are characterized for operation from 0 °C to 70 °C.

FUNCTION TABLE

INPUTS A THRU H	OUTPUT Y
All inputs H	L
One or more inputs L	H

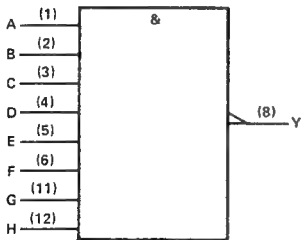
logic diagram



positive logic

$$Y = \overline{A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G \cdot H} \quad \text{or}$$
$$Y = \overline{A} + \overline{B} + \overline{C} + \overline{D} + \overline{E} + \overline{F} + \overline{G} + \overline{H}$$

logic symbol†

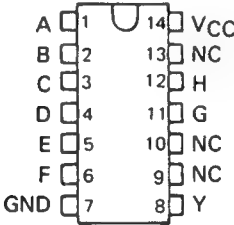


†This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.

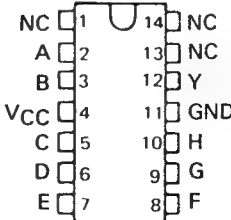
Pin numbers shown are for D, J, N, and W packages.

SN5430 . . . J PACKAGE  
SN54LS30, SN54S30 . . . J OR W PACKAGE  
SN7430 . . . N PACKAGE  
SN74LS30, SN74S30 . . . D OR N PACKAGE

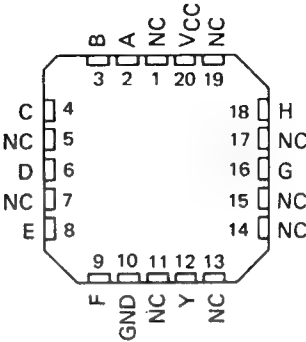
(TOP VIEW)



SN5430 . . . W PACKAGE  
(TOP VIEW)



SN54LS30, SN54S30 . . . FK PACKAGE  
(TOP VIEW)



NC - No internal connection

FIGURE C-4

# **SN5432, SN54LS32, SN54S32, SN7432, SN74LS32, SN74S32** **QUADRUPL 2-INPUT POSITIVE-OR GATES**

DECEMBER 1983—REVISED MARCH 1988

- Package Options include Plastic "Small Outline" Packages, Ceramic Chip Carriers and Flat Packages, and Plastic and Ceramic DIPs
- Dependable Texas Instruments Quality and Reliability

## description

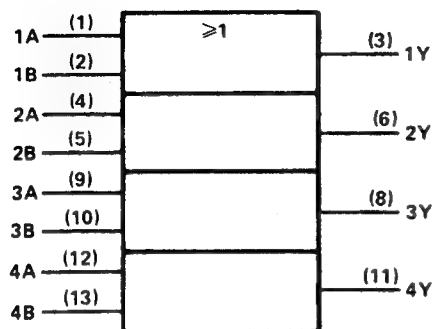
These devices contain four independent 2-input OR gates.

The SN5432, SN54LS32 and SN54S32 are characterized for operation over the full military range of  $-55^{\circ}\text{C}$  to  $125^{\circ}\text{C}$ . The SN7432, SN74LS32 and SN74S32 are characterized for operation from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .

FUNCTION TABLE (each gate)

INPUTS		OUTPUT
A	B	Y
H	X	H
X	H	H
L	L	L

## logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.

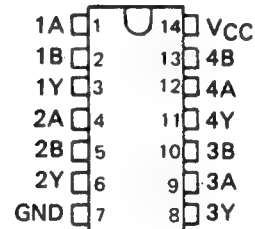
Pin numbers shown are for D, J, N, or W packages.

SN5432, SN54LS32, SN54S32 . . . J OR W PACKAGE

SN7432 . . . N PACKAGE

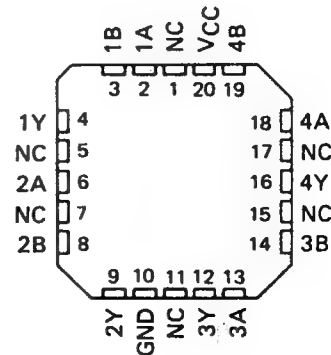
SN74LS32, SN74S32 . . . D OR N PACKAGE

(TOP VIEW)



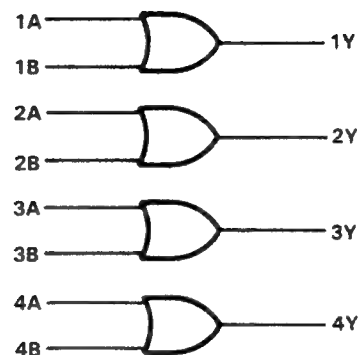
SN54LS32, SN54S32 . . . FK PACKAGE

(TOP VIEW)



NC - No internal connection

## logic diagram



## positive logic

$$Y = A + B \text{ or } Y = \overline{\overline{A} \cdot \overline{B}}$$

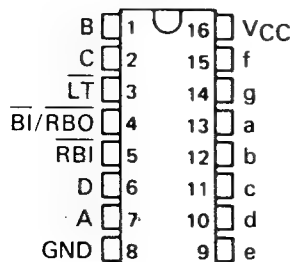
FIGURE C-5

# SN5446A, '47A, '48, SN54LS47, 'LS48, 'LS49, SN7446A, '47A, '48, SN74LS47, 'LS48, 'LS49 BCD-TO-SEVEN-SEGMENT DECODERS/DRIVERS

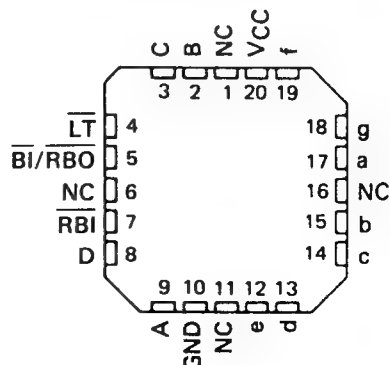
MARCH 1974—REVISED MARCH 1988

'46A, '47A, 'LS47 feature	'48, 'LS48 feature	'LS49 feature
<ul style="list-style-type: none"> <li>• Open-Collector Outputs Drive Indicators Directly</li> <li>• Lamp-Test Provision</li> <li>• Leading/Trailing Zero Suppression</li> </ul>	<ul style="list-style-type: none"> <li>• Internal Pull-Ups Eliminate Need for External Resistors</li> <li>• Lamp-Test Provision</li> <li>• Leading/Trailing Zero Suppression</li> </ul>	<ul style="list-style-type: none"> <li>• Open-Collector Outputs</li> <li>• Blanking Input</li> </ul>

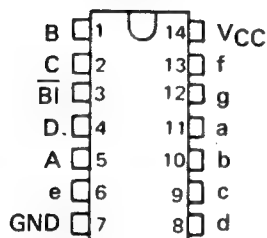
SN5446A, SN5447A, SN54LS47, SN5448,  
SN54LS48 . . . J PACKAGE  
SN7446A, SN7447A,  
SN7448 . . . N PACKAGE  
SN74LS47, SN74LS48 . . . D OR N PACKAGE  
(TOP VIEW)



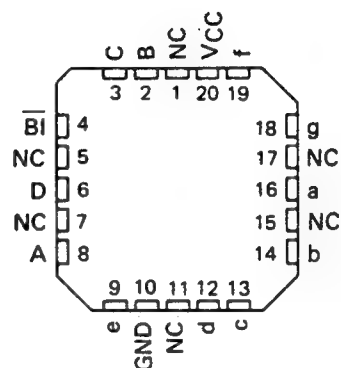
SN54LS47, SN54LS48 . . . FK PACKAGE  
(TOP VIEW)



SN54LS49 . . . J OR W PACKAGE  
SN74LS49 . . . D OR N PACKAGE  
(TOP VIEW)



SN54LS49 . . . FK PACKAGE  
(TOP VIEW)



NC — No internal connection



FIGURE C-6

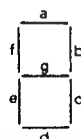
# SN5446A, '47A, '48, SN54LS47, 'LS48, 'LS49, SN7446A, '47A, '48, SN74LS47, 'LS48, 'LS49 BCD-TO-SEVEN-SEGMENT DECODERS/DRIVERS

## description

The '46A, '47A, and 'LS47 feature active-low outputs designed for driving common-anode LEDs or incandescent indicators directly. The '48, 'LS48, and 'LS49 feature active-high outputs for driving lamp buffers or common-cathode LEDs. All of the circuits except 'LS49 have full ripple-blanking input/output controls and a lamp test input. The 'LS49 circuit incorporates a direct blanking input. Segment identification and resultant displays are shown below. Display patterns for BCD input counts above 9 are unique symbols to authenticate input conditions.

The '46A, '47A, '48, 'LS47, and 'LS48 circuits incorporate automatic leading and/or trailing-edge zero-blanking control ( $\overline{\text{RBI}}$  and  $\overline{\text{RBO}}$ ). Lamp test ( $\overline{\text{LT}}$ ) of these types may be performed at any time when the  $\overline{\text{BI}}/\overline{\text{RBO}}$  node is at a high level. All types (including the '49 and 'LS49) contain an overriding blanking input ( $\overline{\text{BI}}$ ), which can be used to control the lamp intensity by pulsing or to inhibit the outputs. Inputs and outputs are entirely compatible for use with TTL logic outputs.

The SN54246/SN74246 and '247 and the SN54LS247/SN74LS247 and 'LS248 compose the  and the  with tails and were designed to offer the designer a choice between two indicator fonts.



SEGMENT  
IDENTIFICATION



NUMERICAL DESIGNATIONS AND RESULTANT DISPLAYS

'46A, '47A, 'LS47 FUNCTION TABLE (T1)

DECIMAL OR FUNCTION	INPUTS						$\overline{\text{BI}}/\overline{\text{RBO}}^\dagger$	OUTPUTS							NOTE
	$\overline{\text{LT}}$	$\overline{\text{RBI}}$	D	C	B	A		a	b	c	d	e	f	g	
0	H	H	L	L	L	L	H	ON	ON	ON	ON	ON	ON	OFF	1
1	H	X	L	L	L	H	H	OFF	ON	ON	OFF	OFF	OFF	OFF	
2	H	X	L	L	H	L	H	ON	ON	OFF	ON	ON	OFF	ON	
3	H	X	L	L	H	H	H	ON	ON	ON	ON	OFF	OFF	ON	
4	H	X	L	H	L	L	H	OFF	ON	ON	OFF	OFF	ON	ON	
5	H	X	L	H	L	H	H	ON	OFF	ON	ON	OFF	ON	ON	
6	H	X	L	H	H	L	H	OFF	OFF	ON	ON	ON	ON	ON	
7	H	X	L	H	H	H	H	ON	ON	ON	OFF	OFF	OFF	OFF	
8	H	X	H	L	L	L	H	ON	ON	ON	ON	ON	ON	ON	
9	H	X	H	L	L	H	H	ON	ON	ON	OFF	OFF	ON	ON	
10	H	X	H	L	H	L	H	OFF	OFF	OFF	ON	ON	OFF	ON	
11	H	X	H	L	H	H	H	OFF	OFF	ON	ON	OFF	OFF	ON	
12	H	X	H	H	L	L	H	OFF	ON	OFF	OFF	OFF	ON	ON	
13	H	X	H	H	L	H	H	ON	OFF	OFF	ON	OFF	ON	ON	
14	H	X	H	H	H	L	H	OFF	OFF	OFF	ON	ON	ON	ON	
15	H	X	H	H	H	H	H	OFF	OFF	OFF	OFF	OFF	OFF	OFF	
$\overline{\text{BI}}$	X	X	X	X	X	X	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF	2
$\overline{\text{RBI}}$	H	L	L	L	L	L	L	OFF	OFF	OFF	OFF	OFF	OFF	OFF	3
$\overline{\text{LT}}$	L	X	X	X	X	X	H	ON	ON	ON	ON	ON	ON	ON	4

H = high level, L = low level, X = irrelevant

- NOTES: 1. The blanking input ( $\overline{\text{BI}}$ ) must be open or held at a high logic level when output functions 0 through 15 are desired. The ripple-blanking input ( $\overline{\text{RBI}}$ ) must be open or high if blanking of a decimal zero is not desired.
2. When a low logic level is applied directly to the blanking input ( $\overline{\text{BI}}$ ), all segment outputs are off regardless of the level of any other input.
3. When ripple-blanking input ( $\overline{\text{RBI}}$ ) and inputs A, B, C, and D are at a low level with the lamp test input high, all segment outputs go off and the ripple-blanking output ( $\overline{\text{RBO}}$ ) goes to a low level (response condition).
4. When the blanking input/ripple blanking output ( $\overline{\text{BI}}/\overline{\text{RBO}}$ ) is open or held high and a low is applied to the lamp-test input, all segment outputs are on.

$^\dagger \overline{\text{BI}}/\overline{\text{RBO}}$  is wire AND logic serving as blanking input ( $\overline{\text{BI}}$ ) and/or ripple-blanking output ( $\overline{\text{RBO}}$ ).

FIGURE C-7

# **SN5474, SN54LS74A, SN54S74, SN7474, SN74LS74A, SN74S74** **DUAL D-TYPE POSITIVE-EDGE-TRIGGERED FLIP-FLOPS WITH PRESET AND CLEAR**

DECEMBER 1983 — REVISED MARCH 1988

- Package Options Include Plastic "Small Outline" Packages, Ceramic Chip Carriers and Flat Packages, and Plastic and Ceramic DIPs
- Dependable Texas Instruments Quality and Reliability

## description

These devices contain two independent D-type positive-edge-triggered flip-flops. A low level at the preset or clear inputs sets or resets the outputs regardless of the levels of the other inputs. When preset and clear are inactive (high), data at the D input meeting the setup time requirements are transferred to the outputs on the positive-going edge of the clock pulse. Clock triggering occurs at a voltage level and is not directly related to the rise time of the clock pulse. Following the hold time interval, data at the D input may be changed without affecting the levels at the outputs.

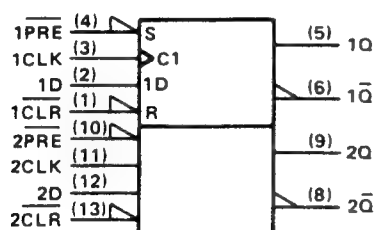
The SN54' family is characterized for operation over the full military temperature range of  $-55^{\circ}\text{C}$  to  $125^{\circ}\text{C}$ . The SN74' family is characterized for operation from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .

FUNCTION TABLE

INPUTS				OUTPUTS	
PRE	CLR	CLK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	$H^{\dagger}$	$H^{\dagger}$
H	H	$\uparrow$	H	H	L
H	H	$\uparrow$	L	L	H
H	H	L	X	$Q_0$	$\bar{Q}_0$

$\dagger$  The output levels in this configuration are not guaranteed to meet the minimum levels in  $V_{OH}$  if the lows at preset and clear are near  $V_{IL}$  maximum. Furthermore, this configuration is nonstable; that is, it will not persist when either preset or clear returns to its inactive (high) level.

## logic symbol<sup>‡</sup>

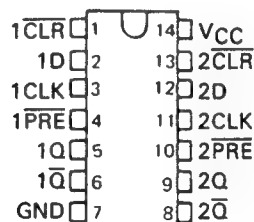


<sup>‡</sup>This symbol is in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.

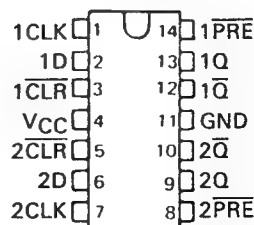
Pin numbers shown are for D, J, N, and W packages.

SN5474 . . . J PACKAGE  
SN54LS74A, SN54S74 . . . J OR W PACKAGE  
SN7474 . . . N PACKAGE  
SN74LS74A, SN74S74 . . . D OR N PACKAGE

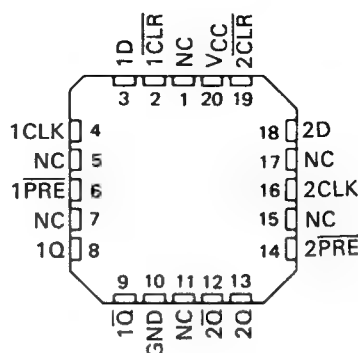
(TOP VIEW)



SN5474 . . . W PACKAGE  
(TOP VIEW)



SN54LS74A, SN54S74 . . . FK PACKAGE  
(TOP VIEW)



NC - No internal connection

## logic diagram (positive logic)

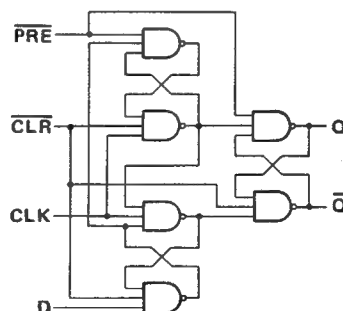


FIGURE C-8



# SN54LS139A, SN54S139, SN74LS139A, SN74S139A DUAL 2-LINE TO 4-LINE DECODERS/DEMULTIPLEXERS

DECEMBER 1972—REVISED MARCH 1988

- Designed Specifically for High-Speed:  
Memory Decoders  
Data Transmission Systems
- Two Fully Independent 2- to 4-Line  
Decoders/Demultiplexers
- Schottky Clamped for High Performance

## description

These Schottky-clamped TTL MSI circuits are designed to be used in high-performance memory-decoding or data-routing applications requiring very short propagation delay times. In high-performance memory systems, these decoders can be used to minimize the effects of system decoding. When employed with high-speed memories utilizing a fast-enable circuit, the delay times of these decoders and the enable time of the memory are usually less than the typical access time of the memory. This means that the effective system delay introduced by the Schottky-clamped system decoder is negligible.

The circuit comprises two individual two-line to four-line decoders in a single package. The active-low enable input can be used as a data line in demultiplexing applications.

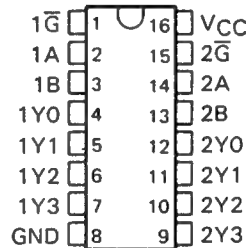
All of these decoders/demultiplexers feature fully buffered inputs, each of which represents only one normalized load to its driving circuit. All inputs are clamped with high-performance Schottky diodes to suppress line-ringing and to simplify system design. The SN54LS139A and SN54S139 are characterized for operation range of  $-55^{\circ}\text{C}$  to  $125^{\circ}\text{C}$ . The SN74LS139A and SN74S139A are characterized for operation from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .

FUNCTION TABLE

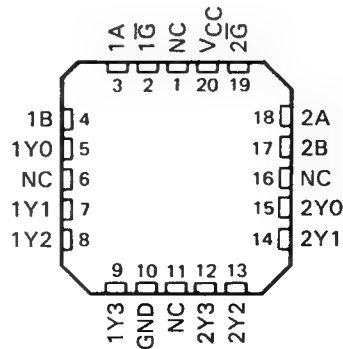
INPUTS			OUTPUTS			
ENABLE	SELECT		Y0	Y1	Y2	Y3
$\bar{G}$	B	A				
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H = high level, L = low level, X = irrelevant

SN54LS139A, SN54S139 . . . J OR W PACKAGE  
 SN74LS139A, SN74S139A . . . D OR N PACKAGE  
 (TOP VIEW)

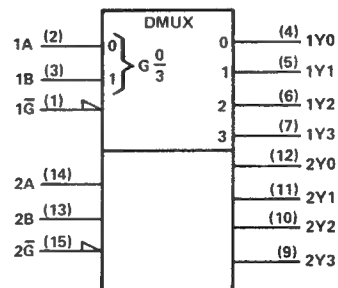
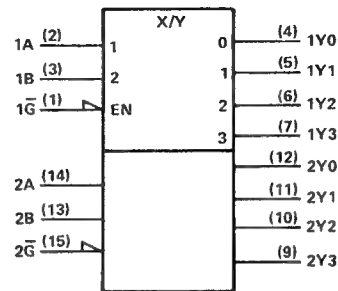


SN54LS139A, SN54S139 . . . FK PACKAGE  
 (TOP VIEW)



NC—No internal connection

## logic symbols (alternatives)<sup>†</sup>



<sup>†</sup>These symbols are in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 617-12.  
 Pin numbers shown are for D, J, N, and W packages.

FIGURE C-9

# **SN54LS682, SN54LS684, SN54LS685, SN54LS687, SN54LS688, SN74LS682, SN74LS684 THRU SN74LS688 8-BIT MAGNITUDE/IDENTITY COMPARATORS**

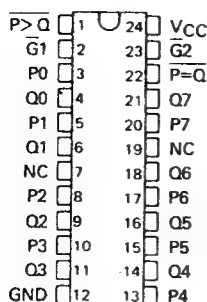
D2617, JANUARY 1981—REVISED MARCH 1988

- Compares Two-8-Bit Words
- Choice of Totem-Pole or Open-Collector Outputs
- Hysteresis at P and Q Inputs
- 'LS682 has 20-k $\Omega$  Pullup Resistors on the Q Inputs
- SN74LS686 and 'LS687 . . . JT and NT 24-Pin, 300-Mil Packages

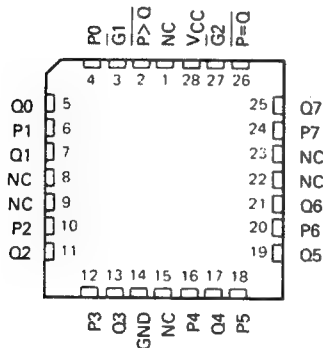
TYPE	$\overline{P} = \overline{Q}$	$\overline{P} > \overline{Q}$	OUTPUT ENABLE	OUTPUT CONFIGURATION	20-k $\Omega$ PULLUP
'LS682	yes	yes	no	totem-pole	yes
'LS684	yes	yes	no	totem-pole	no
'LS685	yes	yes	no	open-collector	no
SN74LS686	yes	yes	yes	totem-pole	no
'LS687	yes	yes	yes	open-collector	no
'LS688	yes	no	yes	totem-pole	no

**SN54LS687 . . . JT PACKAGE  
SN74LS686, SN74LS687 . . . DW OR NT PACKAGE**

(TOP VIEW)

**SN54LS687 . . . FK PACKAGE**

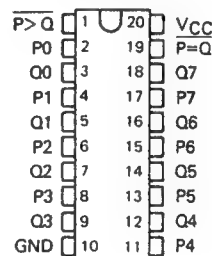
(TOP VIEW)



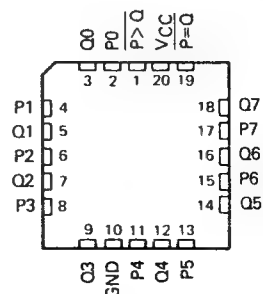
NC—No internal connection

**SN54LS682, SN54LS684, SN54LS685 . . . J PACKAGE  
SN74LS682, SN74LS684, SN74LS685 . . . DW OR N PACKAGE**

(TOP VIEW)

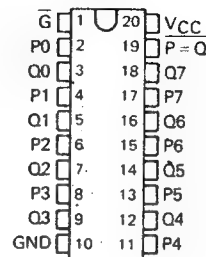
**SN54LS682, SN54LS684, SN54LS685 . . . FK PACKAGE**

(TOP VIEW)

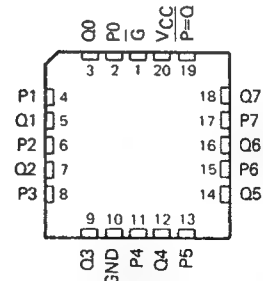


**SN54LS688 . . . J PACKAGE  
SN74LS688 . . . DW OR N PACKAGE**

(TOP VIEW)

**SN54LS688 . . . FK PACKAGE**

(TOP VIEW)

**FIGURE C-10**

# TYPES TIL312 THRU TIL315, TIL327, TIL328, TIL333 THRU TIL335, TIL339 THRU TIL341 NUMERIC DISPLAYS

D1924, SEPTEMBER 1981—REVISED DECEMBER 1982

## SOLID-STATE DISPLAYS WITH RED, GREEN, OR YELLOW CHARACTERS

- 7.62-mm (0.300-inch) Character Height
- Continuous Uniform Segments
- Wide Viewing Angle
- High Contrast
- Yellow and Green Displays are Categorized for Uniformity of Luminous Intensity and Wavelength among Units within Each Category

	SEVEN SEGMENTS WITH RIGHT AND LEFT DECIMALS, COMMON ANODE	SEVEN SEGMENTS WITH RIGHT DECIMAL, COMMON CATHODE	PULSE/MINUS ONE WITH LEFT DECIMAL
RED	TIL312	TIL313	TIL327
GREEN	TIL314	TIL315	TIL328
RED + YELLOW	TIL333	TIL334	TIL335
	TIL339	TIL340	TIL341

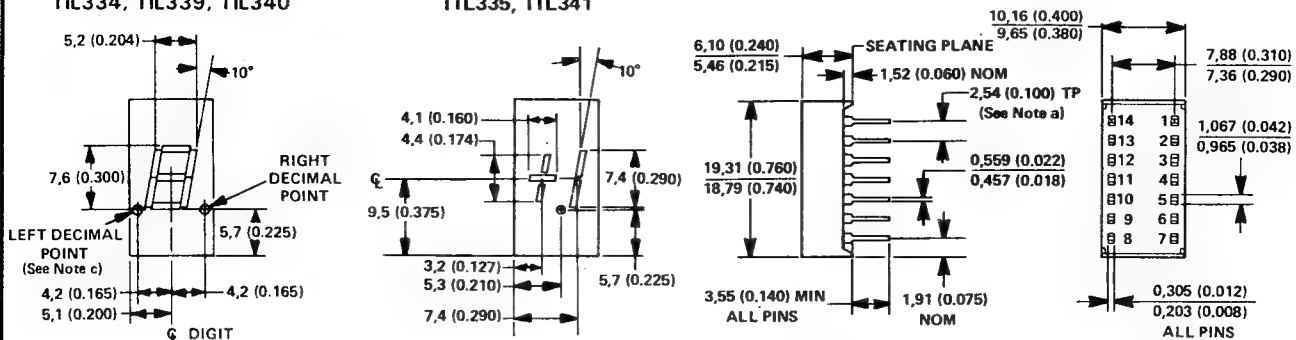
Red + stands for high-efficiency red.

### mechanical data

- NOTES:
- The true position spacing (T.P.) between centerlines is 2.54 mm (0.100 inch). Each pin centerline is within 0.26 mm (0.010 inch) of its true longitudinal position relative to pin 1.
  - All dimensions associated with segments and decimal points are nominal.
  - Left decimal point of TIL313, TIL315, TIL334, and TIL340 is not operational.

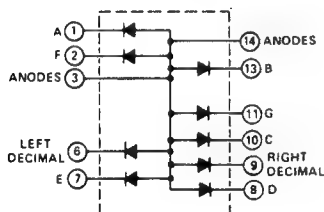
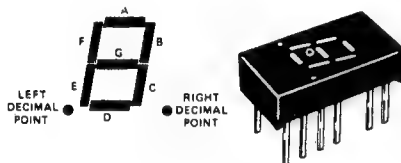
#### TIL312 THRU TIL315, TIL333 TIL334, TIL339, TIL340

#### TIL327, TIL328, TIL335, TIL341



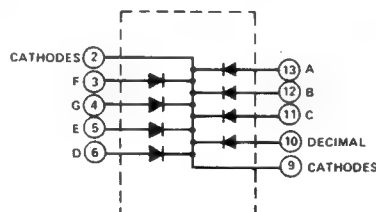
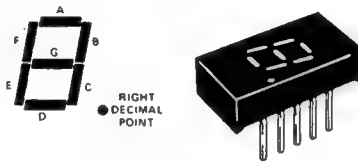
ALL DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

#### TIL312, TIL314, TIL333, TIL339



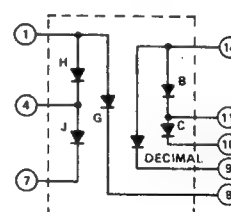
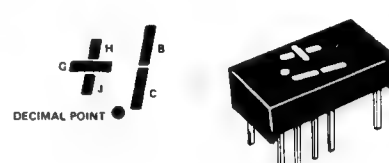
PINS 4, 5, AND 12 OMITTED

#### TIL313, TIL315, TIL334, TIL340



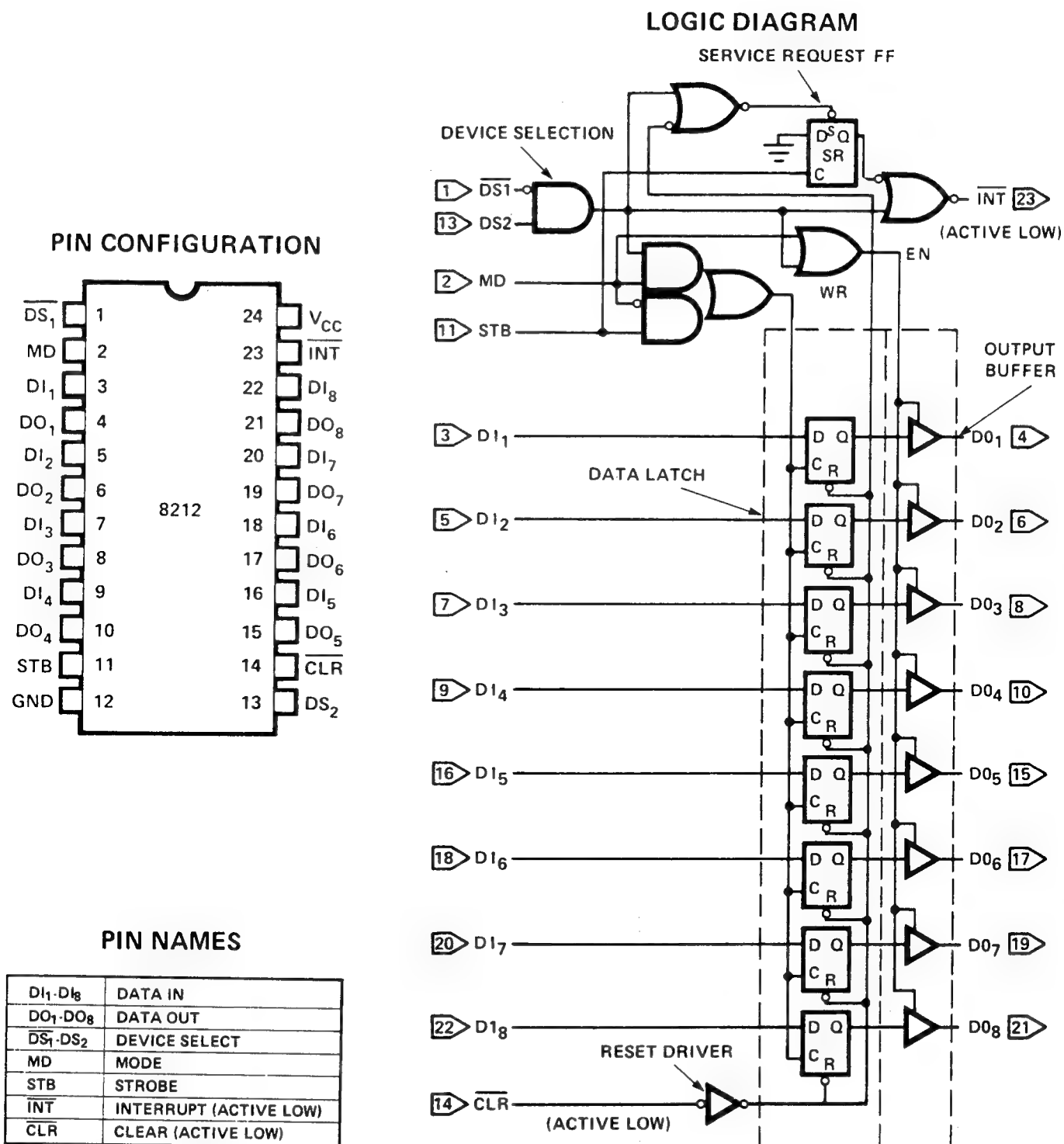
PINS 1, 7, 8, AND 11 OMITTED

#### TIL327, TIL328, TIL335, TIL341



PINS 2, 3, 5, 6, 12, AND 13 OMITTED

FIGURE C-11



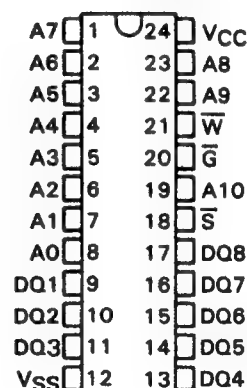
**MOS  
LSI**

**TMS4016  
2048-WORD BY 8-BIT STATIC RAM**

FEBRUARY 1981 – REVISED AUGUST 1983

- 2K X 8 Organization, Common I/O
- Single +5-V Supply
- Fully Static Operation (No Clocks, No Refresh)
- JEDEC Standard Pinout
- 24-Pin 600 Mil (15.2 mm) Package Configuration
- Plug-in Compatible with 16K 5 V EPROMs
- 8-Bit Output for Use in Microprocessor-Based Systems
- 3-State Outputs with  $\bar{S}$  for OR-ties
- $\bar{G}$  Eliminates Need for External Bus Buffers
- All Inputs and Outputs Fully TTL Compatible
- Fanout to Series 74, Series 74S or Series 74LS TTL Loads
- N-Channel Silicon-Gate Technology
- Power Dissipation Under 385 mW Max
- Guaranteed dc Noise Immunity of 400 mV with Standard TTL Loads
- 4 Performance Ranges:

**TMS4016 . . . NL PACKAGE  
(TOP VIEW)**



**PIN NOMENCLATURE**

A0 – A10	Addresses
DQ1 – DQ8	Data In/Data Out
$\bar{G}$	Output Enable
$\bar{S}$	Chip Select
VCC	+ 5-V Supply
VSS	Ground
$\bar{W}$	Write Enable

**ACCESS TIME (MAX)**

TMS4016-12	120 ns
TMS4016-15	150 ns
TMS4016-20	200 ns
TMS4016-25	250 ns

**description**


The TMS4016 static random-access memory is organized as 2048 words of 8 bits each. Fabricated using proven N-channel, silicon-gate MOS technology, the TMS4016 operates at high speeds and draws less power per bit than 4K static RAMs. It is fully compatible with Series 74, 74S, or 74LS TTL. Its static design means that no refresh clocking circuitry is needed and timing requirements are simplified. Access time is equal to cycle time. A chip select control is provided for controlling the flow of data-in and data-out and an output enable function is included in order to eliminate the need for external bus buffers.

Of special importance is that the TMS4016 static RAM has the same standardized pinout as TI's compatible EPROM family. This, along with other compatible features, makes the TMS4016 plug-in compatible with the TMS2516 (or other 16K 5 V EPROMs). Minimal, if any modifications are needed. This allows the microprocessor system designer complete flexibility in partitioning his memory board between read/write and non-volatile storage.

The TMS4016 is offered in the plastic (NL suffix) 24-pin dual-in-line package designed for insertion in mounting hole rows on 600-mil (15.2 mm) centers. It is guaranteed for operation from 0°C to 70°C.

**FIGURE C-13**

**TABLE C-1**  
**Centronics Printer Interface**

SIGNAL PIN NO.	SIGNAL NAME	REMARKS
1	<u>STROBE</u>	Data strobe input from computer
2	DATA 1	 Parallel data from computer
3	DATA 2	
4	DATA 3	
5	DATA 4	
6	DATA 5	
7	DATA 6	
8	DATA 7	
9	DATA 8	
10	<u>ACK</u>	Data acknowledge output from computer
11	BUSY	Busy output from printer
12	PE	Paper empty output from printer
13	SLCT	Selected output from printer
14	=OV	Not used—Not terminated
15	OSCXT	Not used—Not terminated
16	GROUND	Signal ground
17	GROUND	Chassis ground
18	+5V	
19–30		
31	<u>INPUT PRIME</u>	
32	<u>FAULT</u>	
33–36		Not used—Not terminated

INPUT CONNECTOR: The input connector is a 36-pin D connector, AMP champ series (AMP P/N 2-55-2275-1).















Delmar Publishers Inc.™

ISBN 0-8273-5849-0

90000



9 780827 358492